2

This chapter is an excerpt from "SGML CD" by Bob DuCharme (Prentice Hall PTR, Charles F. Goldfarb Series on Open Information Management, ISBN 0-13-475740-8). See www.snee.com/bob/sgmlfree for more information, including links to all the software covered in the book. See also www.snee.com/bob/xmlann for "XML: The Annotated Specification" in the same series.

EDITING SGML DOCUMENTS WITH THE EMACS TEXT EDITOR

Emacs (pronounced "ee-max") started off in 1976 as a series of "editor macros" (hence its name) written by Richard Matthew Stallman for the TECO text editor on the DEC PDP-10 minicomputer. Since becoming its own program separate from TECO, it has become extremely popular and widely used for two main reasons: first, free versions of Emacs are available for nearly every computer in existence; second, it's completely customizable.

Many text editors and word processors claim to be "completely customizable." Some let you reassign each key's purpose, and they let you assign a series of operations to be performed by one or two keystrokes or menu choices. Perhaps this series of operations can have loops of repeated statements and "if" statements that execute one or another group of instructions based on whether a particular condition is true. If so, the editor's proponents claim that its macro facility features a "full-fledged programming language." This is usually an exaggeration, but not with Emacs.

Emacs's accompanying programming language is known as "Emacs LISP" because it's based on LISP, a grand old programming language that first gained fame for its use in artificial intelligence work. (The connection between a text editor and artificial intelligence? Manipulating text.) You don't need to learn Emacs LISP to benefit from it; many Emacs users have made their Emacs LISP programs available to anyone who wants to run them, whether these other users understand the syntax used to create them or not.

One collection of Emacs LISP programs called PSGML can read and parse a DTD and document instance well enough to turn Emacs into a menu-driven SGML editor. PSGML can:

- Insert required elements automatically.
- Help you find tagging mistakes.
- Display tags, data content, comments, and entity references in different fonts or colors, making data content easier to read and markup easier to find.
- Let you enter and edit element attributes using a form with helpful information about declared and default values.
- Enable access to all its important functions (and, when inserting tags, lists of valid element types) on pull-down menus when run on systems that support them.

And, thanks to its author Lennart Staflin, PSGML is as free as Emacs!

In this chapter, we'll first learn the basics of using Emacs with any text file. Then, we'll learn how to install PSGML and use it to edit SGML documents.

<TIP> **Emacs's venerable age means that various terms may seem odd to users accustomed to the terminology of current big-selling software. This chapter explains the Emacs terms and the equivalent terms on currently popular word processors and editors when possible so that you'll more easily understand the vocabulary used in Emacs's on-line help and other available literature.**

Getting Emacs

Emacs's age and the free availability of its source code have led to different versions being available. The standard-bearer at any given time is the Free Software Foundation's GNU Emacs; the further an Emacs implementation deviates from the GNU version, the greater the chance that PSGML won't work with it. I used the DOS EMX release 19.29.2 version of GNU Emacs and the 19.30 Windows 95 version of GNU Emacs while testing this chapter, and I used version 1.0 alpha 6 of PSGML.

All versions of PSGML work with release 19.19 or later of GNU Emacs and release 19.9 or later of the XWindows offshoot of GNU known as Xemacs (formerly Lucid Emacs). As I write this, the most recent version of GNU Emacs is 19.34.

<TIP> **Staflin developed PSGML on a UNIX system, so if you have problems running the latest version under your operating system, try another release of PSGML. He makes several releases available.**

Editing Text Files with Emacs

Emacs often gives you a choice of several keystrokes and commands to perform certain operations. This section covers the minimum amount necessary to get by. Keep in mind that many procedures described here have alternatives that you may prefer.

Starting and Quitting Emacs

To start up Emacs, simply enter

```
emacs
```

at your operating system prompt. In a windowed environment, you can set an icon to perform a similar command.

When Emacs is started with no filename as a parameter, it often displays information about your version of Emacs (see Fig. 2.1).

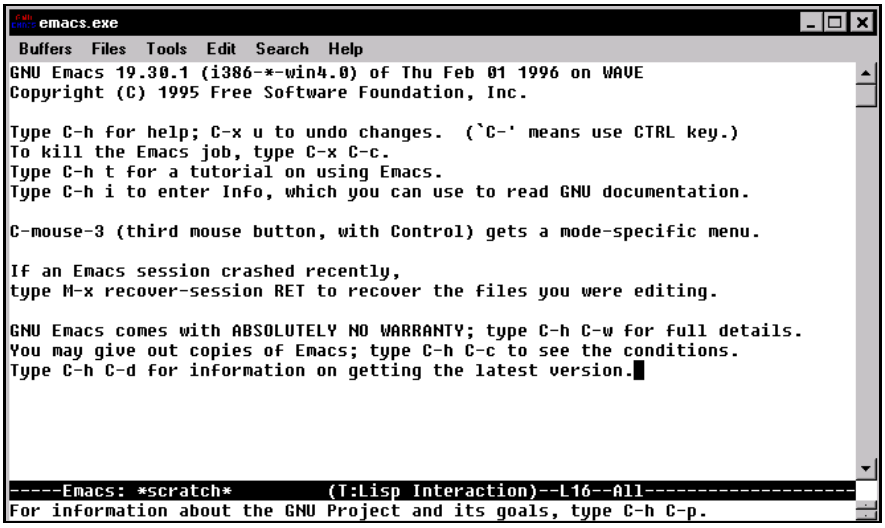

Figure 2.1. Opening Emacs screen in a windowed environment

The bottom line, which begins “For information about...” in the illustration, is known as the “minibuffer.“ You and Emacs enter information there to provide details about what each of you is doing (or trying to do). Emacs puts messages there to tell you things like the number of replacements a search-and-replace operation performed; you use the minibuffer as a command line when necessary and enter text in response to any questions Emacs asks you—for example, whether to save an edited file when quitting.

Just above the minibuffer is the mode line, which tells you the name of the buffer or file you're editing and other details about it. It tells you, as a percentage of the file's total number of lines, how far down in the file the displayed part is. For example, if your cursor is two-thirds of the way down, it says “66%.”

The mode line also tells you, in parentheses, which major mode the current buffer is in. Emacs major modes are states in which certain commands are available and certain settings have particular values. Minor modes generally describe a specific setting; for example, overwrite mode is a minor mode.

Customized major modes are available to edit the text of various programming languages and other kinds of text. In fact, this chapter covers one of these major modes: the PSGML mode.

When starting up Emacs, adding a filename to the `emacs` command, like this,

```
emacs myfile.txt
```

tells Emacs to edit that file, whether it exists or not. If it doesn't exist, Emacs creates an empty buffer with that file's name and, the first time you tell Emacs to save the file, it creates that file for you.

But what's a buffer? It's an area in the computer's memory set aside for each document. Nearly all text editors, word processors, and even spreadsheet programs set aside an area in

memory to work on a copy of each document that you edit. This is why you lose your work if the computer loses power or a program stops running before you save changes—you're making changes to a copy of your document in the computer's memory.

Although many programs use the concept of buffers, Emacs uses the word more than most, so familiarity with the term makes it easier to understand Emacs's on-line help and error messages. Emacs buffers with files in them are usually named after those files.

To end an Emacs session enter **C-x C-c** (**Ctrl+X** followed by **Ctrl+C**—the next section further describes Emacs's notation for expressing combination keystrokes). If your version of Emacs has menus, select **Exit Emacs** from the **Files** menu. If you have no unsaved changes, Emacs returns you to your operating system prompt or closes Emacs's window if you're using a windowed operating system. If you have buffers with unsaved changes, Emacs asks you the following for each one, filling in the appropriate path and filenames:

```
Save file c:/pathname/filename.ext? (y, n, !, ., q, C-r or C-h)
```

It's asking "do you really want to leave Emacs, even though you have copies of documents in memory with changes you haven't saved onto the disk?" If you made changes that you don't want to save, enter **n** to indicate "no." Enter **y** to indicate "yes," and **C-h** to invoke the on-line help, which describes the other possible answers to this prompt.

If you answer **n** to any "Save file?" query, Emacs gives you one more chance to recant before finally quitting:

```
Modified buffers exist; exit anyway? (yes or no)
```

Emacs Commands

Some versions of Emacs offer pull-down menus that let you take advantage of most of its features, and PSGML adds several new menus to Emacs's regular ones. When entering text and moving your cursor around, menus are often slower than keystroke commands, but they're handy for little-used features whose keystrokes are difficult to remember. This chapter mentions each command's corresponding menu choice when there is one.

Usually, you tell Emacs what to do with a special keystroke. For example, **C-a** (pronounced "Control A"—shorthand for pressing your **Ctrl** key and the letter "A" together) means "jump the cursor to the beginning of the current line."

You execute many Emacs commands with a key known as the "meta" key. Some early computers had a special key with this label on it; it was used in combination with other keys just as the **Ctrl** key is. With keyboards that have no "Meta" key (which includes every keyboard I've ever seen) you use the **Escape** or **Alt** keys.

A metakey combination is often written with an "M-" preceding the other key. For example, **M-f** refers to pressing the **Escape** followed by the **f**. Note that, unlike the **Ctrl** key in the **C-f** key combination, you press the **f** after pressing **Escape**, not while pressing it. If your computer and version of Emacs let you use the **Alt** key for metakeys, press it with the other key just as you would press the **Ctrl** or **Shift** key simultaneously with another key.

Odd commands actually exist for which you press **Escape** followed by a **Ctrl** key combination. Instead of writing this as **M-C-x** for meta-control-x, Emacs documentation mercifully refers to such a combination as **ESC C-x**. (If your version of Emacs lets you use **Alt** as the metakey, you can press **Alt+Ctrl+x**.) This chapter only describes one such keystroke.

Many Emacs commands require a pair of keystrokes. Either or both of these may be a combination keystroke; for example, **C-x** followed by **C-s** (more commonly written as **C-x C-s**) saves the currently displayed file. Combination keystrokes are often grouped so that all those with the same first keystroke belong to a similar category of commands. For example, all keystrokes beginning with **C-x** involve opening and saving files or quitting Emacs, much like the choices on the **File** menu of most windowed operating systems' applications.

Sometimes the first of a pair of keystrokes displays a menu of potential second keystrokes. For example, pressing **C-h** displays the following in the minibuffer:

```
C-h (Type ? for further options)
```

(As we'll see in the "Available On-line Help" section, pressing the **?** key then gives you a description of your options.) PSGML makes good use of this type of prompt so that you don't need to memorize many keystrokes.

Emacs can do so much that there are not enough keystrokes to go around, so it offers a command line where you can enter commands. Press **M-x** (**Escape** followed by the **x** key) to display this command line in the minibuffer. This minibuffer command line is important to an add-in package such as PSGML that adds customized features to Emacs for the benefit of a specialized audience.

For example, the `goto-line` command has no keystroke to invoke it—we'll see in the section "Customizing Emacs" how to assign one—but you can jump the cursor to a specific line number by invoking the minibuffer command line with **M-x**, entering `goto-line` in the minibuffer, and pressing **Enter**. When Emacs prompts you with

```
Goto line:
```

at the minibuffer, enter the line number and press **Enter**.

With all of these multi-step command possibilities, it's good to know that a keystroke is available to abort one that you didn't mean to start: **C-g**. For example, if you press **C-x**, Emacs displays the following in the minibuffer to show that it's waiting for you to finish what you started:

```
C-x-
```

If you meant to press **C-s** instead of **C-x**, you couldn't press it now because **C-x C-s** is a very different Emacs command from **C-s.**

If you actually follow through on a command that you didn't mean to execute, pressing **C-_** (or selecting **Undo** from the **Edit** menu) undoes the most recent command.

There's one more common trouble spot to be aware of. Heavy use of the **Escape** key means occasionally pressing it twice in a row by accident. This invokes a command so esoteric that some UNIX versions of Emacs display the following:

```
You have typed ESC ESC, invoking disabled command eval-expression:
Evaluate EXPRESSION and print value in minibuffer.
Value is also consed on to front of the variable 'values'.

You can now type
Space to try the command just this once,
      but leave it disabled,
Y to try it and enable it (no questions if you use it again),
N to do nothing (command remains disabled).
```

Emacs is telling you "You just invoked an esoteric disabled command that you probably didn't mean to use. Do you really want to go through with this?" Enter **n.** Entering **Esc Esc** using other versions of Emacs just shows

```
ESC ESC
```

in the minibuffer; enter **C-g** to abort this.

Moving Your Cursor Around

Most versions of Emacs support the regular cursor up, down, left, and right keys, **C-Left** and **C-Right** to move a whole word left or right, and the **Page Up** and **Page Down** keys for scrolling a screenful of text at a time. If you have problems with these keys—for example, if you use Emacs on a UNIX system from a PC over a phone line and your telecommunications program treats **Page Down** as the "start downloading a file" command—you can use the original Emacs cursor movement keys from the days before computer keyboards had any cursor or **Page Up** and **Page Down** keys. The following list includes the Emacs commands invoked by these keystrokes because on-line help and Emacs documentation sometimes refer to them by these names instead of the keystrokes.

C-p	cursor up	`previous-line`
C-n	cursor down	`next-line`
C-b	cursor left	`backward-character`
C-f	cursor right	`forward-character`
M-b	word left	`previous-word`
M-f	word right	`next-word`
M-v	page up	`previous-page`
C-v	page down	`next-page`

The following keystrokes are also handy for getting around quickly:

C-a	beginning of current line	`beginning-of-line`

C-e	end of current line	`end-of-line`
M-<	beginning of file in current buffer	`beginning-of-buffer`
M->	end of file in current buffer	`end-of-buffer`

Editing and Deleting Text

To enter text, simply move your cursor to the place where you want the new text and start typing. If you create a brand new document, the beginning of that document is the only place to enter it.

If you type to the end of a line and continue typing, you may see a backslash ("\") appear as the last character of the line, and then your typed text appears on the second line:

```
Returning to the Spouter-Inn from the Chapel, I found Queequeg there quite alon\
e; he having left the Chapel before the benediction some time.
```

The backslash shows that your text isn't really continuing on the second line; the line is too long to appear on one line of your screen (or window), so Emacs just wraps its appearance so that you can see what you're typing.

Of course, you can press **Enter** at any time to start a new line, but any decent word processor gives you the option of letting it automatically insert a carriage return at the last possible place before the cursor reaches the right margin. If you want Emacs to do this, enter **M-x** to bring up its command prompt and enter `auto-fill-mode`. This turns on the "Fill" minor mode; the word "Fill" appears on the mode line to let you know that it's on (see Fig. 2.2).

Figure 2.2. Mode line showing that fill mode is on

What if you're a programmer and you don't want your text editor to combine your program code into paragraphs? How do you turn fill mode off? The same way you turn it on: by entering `auto-fill-mode` at the **`M-x`** prompt. This command is known as a toggle because entering it when the mode is on turns it off, and entering it when the mode is off turns it on.

To reset the right margin (or, in Emacs parlance, the "fill column"), enter **`C-u`** followed by a number indicating the desired width of the text and then **`C-x f`**.

To delete the character at the cursor, the **`Delete`** key should work, but if not, **`C-d`** does the same thing. Your **`Backspace`** key should also work.

<TIP> **When using a UNIX system, especially from a PC running a terminal emulation program, the `Delete` key is often the one that deletes the character to the left of the cursor, and `C-d` is your only option for deleting the character at the cursor. Or, `Backspace` deletes the character before the cursor and the `Delete` key does nothing, so that `C-d` is still your only option for deleting the character at the cursor. I'll refer to the key that deletes the character at the cursor as the `Delete` key and the one that deletes the character just before the cursor as the `Backspace` key, regardless of which keys on your keyboard perform these functions.**

To delete from the cursor position to the end of the line, press **`C-k`** (`kill-line`, a slightly misleading command name because it kills the line from the cursor position to the end, not the entire line).

One great feature of Emacs is its ability to re-justify paragraphs. For example, imagine that you typed a paragraph of text but then deleted a short sentence in the third line of that paragraph, and now you have a big gap in your right margin:

```
He made me a present of his embalmed head; took out his enormous
tobacco wallet, and groping under the tobacco, drew out some thirty
dollars in silver; then spreading them
on the table, and mechanically dividing them into two equal portions,
pushed one of them towards me, and said it was mine.
```

With your cursor at any point in that paragraph, press **M-q** (`fill-paragraph`) or select **Fill** from the **Edit** menu to adjust the paragraph to look like this:

```
He made me a present of his embalmed head; took out his enormous
tobacco wallet, and groping under the tobacco, drew out some thirty
dollars in silver; then spreading them on the table, and mechanically
dividing them into two equal portions, pushed one of them towards me,
and said it was mine.
```

This is a typical feature in word processors, but not in text editors, most of which are designed for programmers who deal in lines instead of paragraphs. `fill-paragraph` combines all the text between the last blank line before the cursor and the next blank line after the cursor into a new paragraph. If Emacs combines more text into a paragraph than you wanted, you can undo it with the **C-**_ undo command.

A handy feature for adding a large amount of text is the ability to insert an existing text file's contents at the cursor location. To do this, press **C-x C-i** and enter the filename at the minibuffer prompt:

```
Insert file: c:\pathname\
```

If you don't want to insert a file from the current directory, backspace over the pathname displayed and enter the appropriate one.

Saving Edits

To save the file you're currently editing with the name already assigned to it, enter **`C-x C-s`** (or select **Save Buffer** from the **Files** menu) to invoke the `save-buffer` command. To save it with a new name, enter **`C-x C-w`** (or select **Save Buffer As...** from the **Files** menu) to invoke the `write-file` command. Emacs will ask you to enter the new name. If you like, you can backspace over the displayed path name and enter a new one:

```
Write file: c:/pathname/
```

Copying, Moving, and Deleting Blocks of Text

Emacs lets you copy, move, and delete user-defined blocks of text with operations similar to the cutting, copying, and pasting of most popular word processors and text editors. However, its roots in the bronze age of computers mean that the vocabulary describing these operations may seem strange to the modern user, so a brief glossary is helpful:

region	A block of text.
point	The cursor's location.
mark	If before the point, the start of a region; if after, the end of a region.
kill ring	A temporary region of memory that stores text to move or copy later on. On more modern word processors this is often called the "clipboard," although a kill ring can hold multiple blocks of cut or copied text.
kill-region	The act of deleting a marked region and saving the text in a temporary buffer in the kill ring. On modern word processors this is known as cutting a block into the clipboard.

copy-region Copying the marked region into the kill ring.

yank Copying the kill ring's most recent addition to the cursor location. Word processors using the clipboard metaphor call this "pasting" the clipboard's contents.

Choices on the **Edit** menu that implement these features often use the more modern terminology.

You begin a copy, move, or delete operation by indicating the block of text in question. With your cursor at the beginning of the text, press **C-space** (or **C-@**, both for the `set-mark-command`) to set the mark and then move your cursor to the end of the region. Or, start with one of these two keystrokes at the end of the region and move your cursor to the beginning.

Some versions of Emacs highlight the currently selected region, but most don't, so it's easy to forget which text is selected. This can lead to trouble with the delete command, so the `exchange-point-and-mark` command (**C-x C-x**) makes it easy to check the region's current boundaries by jumping the cursor to the currently set mark and setting the point as the new mark. Press the double key combination again to return your cursor to its location before you exchanged the point with the mark, and you can then continue where you left off. Essentially, doing this command twice jumps your cursor to the region's other boundary and back again so that you can quickly see where that boundary is.

To delete (or "kill") the region, press **C-w** or select **Cut** from the **Edit** menu. If you deleted a region accidentally, "yank" it out of the kill ring to put it back by pressing **C-y**. If you move your cursor before yanking this text, yanking it moves the text to the cursor's new location. (Another **C-w** removes the recently yanked text if you put it in the wrong place.)

The menu equivalent of the `yank` command offers you a bit more flexibility: choose **Select and Paste** from the **Edit** menu to display a cascade menu of your last few yanked or copied pieces of text. Select one of these and Emacs pastes it at the cursor's position.

You can copy a region to the kill ring by selecting **Copy** from the **Edit** menu or by pressing the `kill-ring-save` keystroke: **`M-w.`** This has no effect on the file you are editing, but once the region's text is in the kill ring, you can yank it all you want to make multiple copies.

With word processors that use the clipboard metaphor, cutting or copying text into the clipboard replaces anything that was already there. If you cut or copy something with the intention of pasting it somewhere and then cut or copy something else before you get around to the planned paste operation (**`C-y`**), you've lost the original block of text. The Emacs `yank-pop` command (**`M-y`**) lets you replace the most recently yanked text with whatever was killed or copied into the kill ring before the text that you just yanked. (The "pop" part of the name is programmer's lingo for the operation performed on the data structure holding the killed and copied text.) Pressing **`M-y`** repeatedly continues to replace the recently yanked text with earlier and earlier versions of killed or copied text; how early you can go depends on the version of Emacs you're using. Once you pop the oldest kill ring item, the next pop inserts the most recent item saved in the kill ring. That's why it's called a "ring."

For example, let's say you want to move the "red" line in the following text after the "blue" line.

```
yellow
red
black
white
blue
green
```

You move your cursor to the beginning of the “red” line and press **C-space** to indicate the beginning of a block. Then, you move your cursor to the beginning of the line below it and press **C-w** to kill that marked line.

```
yellow
black
white
blue
green
```

As you move your cursor toward the “red” line's new location, you realize that you didn't want the “white” line there at all, so you kill it with the same keystrokes.

```
yellow
black
blue
green
```

You continue the cursor's journey to the “red” line's new location and put it at the “g” in “green” because “red” will be inserted before the “green” line and after the “blue” line. You press **C-y** to yank the kill ring text to the cursor's location and reel back in horror as you realize that you pasted the “white” line there, not the “red” line:

```
yellow
black
blue
white
green
```

Before rending your clothing in anguish over your mistake, you remember the `yank-pop` command, which yanks earlier kill ring contents to the cursor location. You press **M-y** and the “red” line replaces the mistakenly yanked “white” line.

```
yellow
black
blue
red
green
```

Searching for Text, Replacing Text

The most common Emacs search is known as an "incremental search." Enter **`C-s`** and start typing your search target at the minibuffer's `I-search:` prompt and notice how the cursor jumps to the first place that matches your search target *as you type each character*. For example, let's say we want to search for the word "hazel" in the following. After the "h" is typed, the cursor immediately jumps to the first character after the first "h" it finds, as shown in Fig. 2.3.

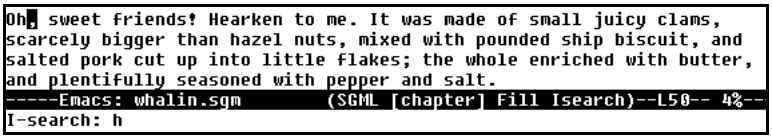

Figure 2.3. Incremental search after the first character is entered

After typing the letter "a," the cursor jumps to the "n" in "than," the first character after the first occurrence of "ha" (see Fig. 2.4).

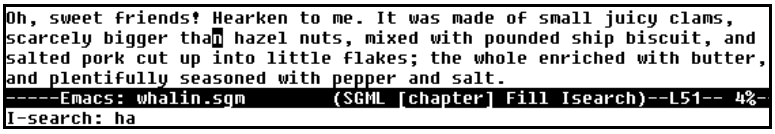

Figure 2.4. Incremental search after second character entered

Type a "z," and the cursor jumps to the "e" after the "haz" in "hazel," as shown in Fig. 2.5.

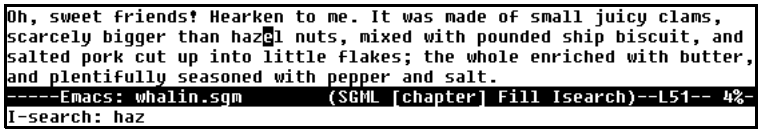

Figure 2.5. Incremental search after third character entered

If you enter a series of characters that Emacs can't find, it tells you in the minibuffer. For example, if you type the letter "x" at the end of the "haz" entered so far, the minibuffer message tells you this:

```
Failing I-search: hazx
```

If you've found the string that you're looking for but want Emacs to look for another occurrence of it, press **C-s** again. When you've done enough searching, press **Enter** to stop the incremental search.

Another handy key when searching is the **Backspace** key. Using it to delete the most recent key in the minibuffer returns the cursor to its most recent search hit. For example, if you backspace over the "z" after Emacs found the word "haz" in the above example, the cursor jumps back to the word "than," which has the first example of the "ha" string currently showing in the minibuffer.

Searching backward is similar to searching forward except that you begin an incremental reverse search with the keystroke **C-r.** For each character you add to the search string, Emacs jumps the cursor to the file's most recent occurrence of the current search string until you either enter a string it can't find or press one of the keystrokes that tells it to stop looking (for example, **Enter**). As with a forward search, the **Backspace** key also deletes the most recent character in the search string, jumping the cursor back to its location before you entered that character.

Emacs has several commands that can perform a search and replace operation. The most versatile is `query-replace`, which you invoke with the **`M-%`** keystroke or the **Query Replace...** choice of the **Search** menu. Press it and Emacs displays the following prompt in the minibuffer:

```
Query replace:
```

Enter the string to replace and press **`Enter`**. For example, let's say you entered "my fault." The minibuffer then asks you to enter the replacement string:

```
Query replace my fault with:
```

Enter the replacement string (we'll change "my fault" to "your fault") and press **`Enter`**. Emacs then tries to find the search string. If it doesn't, it displays the message `Replaced 0 occurrences` in the minibuffer; if it does, it moves the cursor to the first occurrence and displays the following prompt:

```
Query replacing my fault with your fault: (? for help)
```

Enter **`?`** to display a list of options. The following shows the most important ones:

y	Replace this entry and search for the next one. The space bar has the same effect.
n	Don't replace this entry, but skip to the next one.
q	Don't replace this entry. Stop looking.
!	Replace all remaining occurrences of the search string, no questions asked.

Editing Multiple Files

The **C-x C-f** command invokes the `find-file` command. Entering it (or selecting **Find File...** from the **File** menu) tells Emacs "Edit the file whose name I'm about to give you. If it's already in a buffer, show me that buffer; if it's not in a buffer but on a disk, read it into a buffer where I can edit it; if it's not there either, create a buffer that I can treat as a new file." In the minibuffer, Emacs then asks you for the name of the file to edit, substituting your current directory for `pathname`:

```
Find file: c:\pathname\
```

If the file you want is in that directory, just enter its name. Otherwise, edit the pathname by deleting characters with the backspace key or adding new characters before entering the filename. If you can't remember the filename, we'll see how to list the files in the current directory in a separate window and pick one from the list using Emacs's "completion" features described in the next section.

To switch from one active buffer to another, enter **C-x b** for the `switch-to-buffer` command. Emacs prompts you for the buffer's name, which is the same as the name as the file in that buffer. As with `find-file`, you can use the "completion" feature if you don't know the current buffer names.

All buffers share the same kill ring, which makes it easy to copy and move text between them. For example, to move a paragraph from `file1.txt` to `file2.txt`, you would:

1. Mark a region of text in `file1.txt`.
2. Kill it with the **C-w** keystroke.
3. Display the other file by entering **C-x b** and answering the `Switch to buffer:` minibuffer prompt with `file2.txt`.
4. Move your cursor to the insertion point in `file2.txt` with the cursor movement keys or by using **C-s** to search for a string that you know is near the insertion point.

5. Insert the text from the kill ring with **C-y.**

While viewing a particular buffer, pressing **C-x C-s** tells Emacs to save the file in that buffer. It's not unusual when you have several files open in different buffers to save the current one and then try to quit Emacs without saving the others; that's why Emacs displays a reminder if you enter **C-x C-c** to quit when you still have unsaved work.

Completion

When Emacs is waiting for you to enter information and you're not sure what to enter, or you don't feel like typing in all those keystrokes, Emacs can do a lot of the typing for you.

For example, the last section showed you how **C-x C-f** can tell Emacs to open a new buffer for a new file. Let's say you want to edit a file on your hard disk's current directory and you remember that the filename begins with the letters "apr" but can't remember the rest. After you press **C-x C-f** and enter "apr" in the minibuffer, pressing the **Tab** key tells Emacs to fill in as many of the remaining letters as possible. If only one file begins with these letters, it finishes the file's name; if more than one such file exists, Emacs fills in as many as it can. For example, if there are files named `april95.txt` and `april96.txt` in the current directory, and you enter "apr" and press **Tab**, Emacs adds the "il9" because the two files that begin with "apr" both begin with the letters "april9." It also splits the window and lists the files whose names fit this pattern, as shown in Fig. 2.6.

You can then type in the remainder of the name, or, as the new window tells you, you can move your cursor to the name you want and press "Ret" (the **Return** or **Enter** key). We'll see in the next section how to move your cursor from one window to another and how to delete a window when you're done with it.

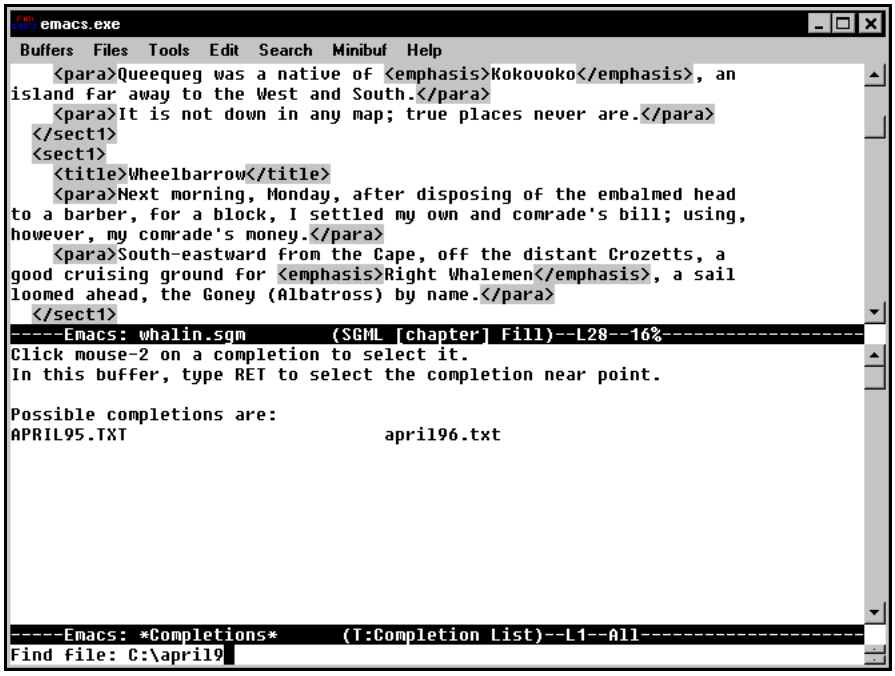

Figure 2.6. Possible completions to the sample `find-file` command

Completion is used for entering more than just filenames. When entering a long Emacs command such as `add-change-log-entry-other-window` at the **`M-x`** prompt, you can type a few characters, press **`Tab`**, type a couple more to make it more unique, press **`Tab`** again, and continue with this process until you have the complete command name. Completion also helps you to enter buffer names after entering **`C-x b`** to switch to a buffer whose name you can't remember, as well as variable names, which we'll learn about in the "Customizing Emacs" section.

Using Multiple Emacs Windows

You can split the Emacs screen into two or more regions known as "windows" to look at multiple buffers at once. (Don't confuse these with the windows used for the operating system

interface in Microsoft Windows, UNIX X-Windows, or the Macintosh—this doesn't create new independent screen rectangles but instead splits the current one.) Sometimes Emacs splits the screen automatically to simultaneously show you the currently active buffer as well as some other information, such as on-line help or a list of potential completions to text you've entered. Even if you have no plans to split the screen into two windows yourself, you should learn the commands that manipulate multiple windows in order to give you better control over the Emacs environment when Emacs splits its default main screen for one of these reasons.

Entering **C-x 2** (or selecting **Split Window** from the **Files** menu) splits the Emacs screen horizontally into two windows. Both windows display the buffer that you were looking at when you pressed **C-x 2**; all commands that we've seen for specifying files and buffers to work on will then apply to the window with the cursor in it.

So, to look at two different files at once, you would:

1. Display one of the files using a method we've already seen.
2. Enter **C-x 2** to split the screen.
3. Enter **C-x C-f** to retrieve the second file into one of the two windows.

When the screen is divided into multiple windows, the keystroke **C-x o** (note that this uses the letter “o,” not the digit zero—as we'll see, **C-x 0** has a very different effect) moves the cursor to the next, or “other,” window. If there is something in the minibuffer, this counts as a little window. So, when viewing Fig. 2.6, which shows the cursor in the minibuffer, pressing **C-x o** moves the cursor to the top window, as shown in Fig. 2.7.

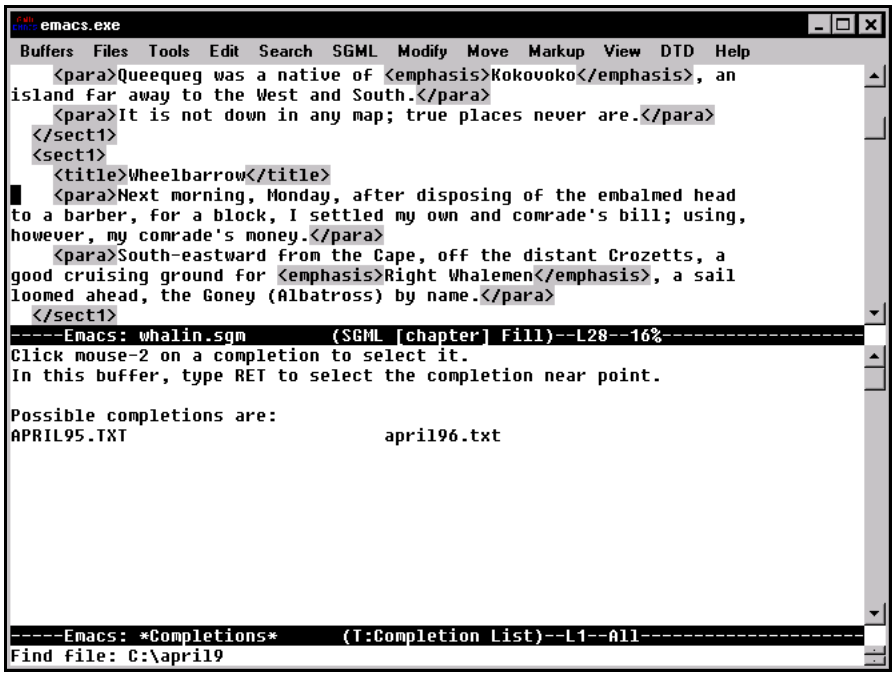

Figure 2.7. Moving the cursor to the next window with the `other-window` command

Pressing it again moves it to the lower window (see Fig. 2.8).

Pressing it a third time moves it back to the minibuffer.

Now we know how to create and navigate among these windows. How do we get rid of them? The **`C-x 0`** keystroke executes the `delete-window` command, which deletes the cursor's current window. (This is the **`C-x zero`** command mentioned earlier, not to be confused with **`C-x o`**.) Think of it as "zeroing out" the cursor's current window.

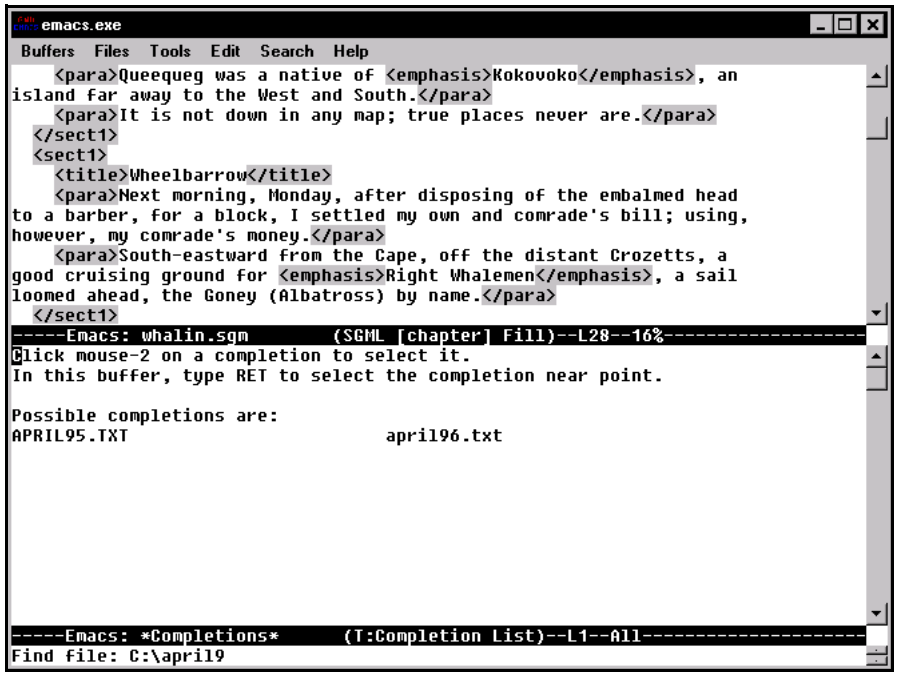

Figure 2.8. Moving to another window with the `other-window` command

The **`C-x 1`** keystroke (and the **One Window** choice of the **Files** menu) executes the `delete-other-windows` command, which expands the cursor's current window and deletes the others. Think of it as "make this window the only one" to help remember it more easily.

If any deleted windows held file buffers, those buffers are still around, they're just not displayed—you can always switch to another with **`C-x b`**.

A similar keystroke is available when you want to switch buffers but don't remember their names: **`C-x C-b`**, the `list-buffers` command. If necessary, this splits the current window and lists the currently existing buffers in the other window. Press **`C-x o`** to move your cursor to this list, move it to the

name of the buffer you want, and press the number **1** to display that buffer in the window where the cursor was when you first pressed **C-x C-b**.

The **Buffers** menu also lists the current buffers; select one to put that buffer in the cursor's window.

Customizing Emacs

Emacs lets you store a series of keystrokes for later playback, which is handy when you're tired of repeatedly pressing the same series of keystrokes. It also lets you customize the environment by writing programs in the Emacs LISP programming language. Even if you've never used LISP or one of its descendants (such as Scheme or CLOS), you can easily make minor changes to other users' programs to gain greater control over your own use of Emacs.

Recording and Executing Macros

Emacs offers a keyboard macro facility that lets you record and play back keystrokes. You begin and end your "recording" of the keystrokes by entering **C-x** followed by one of the parentheses characters. The **C-x (** keystroke, for `start-kbd-macro`, tells Emacs to record the upcoming keystrokes, and **C-x)**, for `end-kbd-macro`, tells Emacs to stop recording.

Pressing **C-x e** executes the `call-last-kbd-macro` command. This executes the macro most recently defined with **C-x (** and **C-x)**.

If you create a macro and want to create another without losing the ability to execute the first one, you can name the first one by pressing **M-x** and entering the `name-last-kbd-macro` command at the minibuffer prompt. After you press `Enter`, Emacs asks you for the macro's name; if you name it `testmacro`, you can then record and use another macro and still invoke `testmacro` the same way you would use any other Emacs command: by entering it at the **M-x** minibuffer prompt.

As soon as you quit Emacs, you lose the most recently named macro and any other named macros saved during that session. What if you want to define macros to use permanently? At the end of this section, after we've covered the basics of the `.emacs` customization file, we'll see how to add named macros so that you can use them in the future.

Emacs LISP

More serious Emacs customization is done by writing Emacs LISP programs. These don't have to be complicated. For example, assigning an Emacs command to a particular keystroke can be done with one line of LISP code.

In the LISP programming language, as with C, everything is defined as a function. You can write your own functions or use collections written by other people known as "packages" to customize Emacs's behavior. PSGML is a LISP package, but before we get to that, we'll start with a simpler collection of LISP functions: the automatic startup functions stored in the `.emacs` (pronounced "dot emacs") file.

<TIP> **Because DOS won't allow filenames that begin with a period, DOS and 16-bit Windows versions of Emacs usually use the name `_emacs` for the file of automatic startup functions.**

While the syntax of the `.emacs` functions may seem odd to someone with no experience in LISP or its descendants, you can write your own LISP functions by merely taking others and changing the crucial parts. For example, let's say I see the following line in someone's `.emacs` file to set **`M-g`** to invoke the `goto-line` command:

```
(global-set-key "\eg" 'goto-line)   ; M-g prompts for line number to enter
```

The `goto-line` command is normally not invoked by (or "bound to," as the Emacs types say) any key, so that you need to enter **`M-x`** and then `goto-line` at the minibuffer command prompt in order to go to a specific line by its number. Programmers and SGML people often run programs that identify problems in program and data files by their line numbers, so they need to use Emacs's `goto-line` command so often that it would be easier to just press a key or two to invoke it.

Copy the `global-set-key` line shown above into your `.emacs` file and quit and restart Emacs. From that point on, pressing **`M-g`** displays a `Goto line:` prompt in the minibuffer, which waits for you to tell Emacs where to put the cursor.

Another command popular with PC word-processor users is `overwrite-mode`, which toggles Emacs between insert and replace modes. In other words, entering this command at the **`M-x`** prompt puts Emacs in replace mode if you were already in insert mode, so that typed characters replace existing characters at the cursor, or it puts you in insert mode if you're already in replace mode, so that newly typed characters move any existing characters on the cursor's right further to the right.

PC users are accustomed to this toggling behavior from their **`Insert`** key and don't necessarily want to bring up a command line and enter a 14-letter command. It would be nice to define a LISP function in the `.emacs` file that assigns this command to the **`M-i`** keystroke. Let's look closer at the line that assigned the `goto-line` command to the **`M-g`** keystroke and see which parts to keep and which we can change in order to create a LISP function that gives **`M-i`** the behavior we want.

LISP functions are usually in the form of a list (the name actually means "List Processor") within parentheses. The first item in the list is the function name; the number and role of the remaining items depend on which function starts the list.

The Emacs LISP `global-set-key` function sets a particular key to perform an Emacs function. (Non-global versions specify key assignments that only work in certain modes.) This takes

two parameters: the key to set and the command to assign to it. You'll rarely assign a command to a single letter or number on your keyboard; as with most Emacs commands, these new commands are invoked by some combination of a letter or a number with the **Escape** or **Ctrl** key.

To assign a key to be pressed after the **Escape** key, precede it with a backslash and the letter "e." For example, we saw that binding **M-g** to invoke the `goto-line` command meant assigning that command to the "\eg" keystroke combination.

Also, remember to put an apostrophe (or, in programmers' parlance, a "single quote") before the command name being assigned to the keystroke. In LISP, this makes the command name a symbol. Don't worry about LISP symbols unless you want to get heavily into LISP; just remember to include it when assigning commands to keystrokes.

One more thing: when Emacs executes LISP code, it ignores a semicolon and anything after it, so use these to insert comments. This makes it easier to remember why you wrote what you did, and it also makes it easier to learn from other Emacs LISP programs out there—assuming the programs have been commented.

So, putting this all together, we can set up the **M-i** keystroke to toggle between insert and replace mode by adding the following line to our `.emacs` file:

```
(global-set-key "\ei" 'overwrite-mode)     ; toggle overwrite mode
```

Assigning a command to a **Ctrl** key combination requires a new Emacs trick: the `quoted-insert` command, entered with **C-q**. This tells Emacs to treat the next character you type as a character to be inserted into the buffer, even if Emacs normally treats it as a command. So, while **C-a** normally means "move my cursor to the beginning of the current line," pressing **C-q C-a** means "insert the C-a character at the cursor posi-

tion." It shows up as ^A, but most programs that do anything with this file treat it very differently from something entered with the "^" character (**`Shift 6`**) followed by the letter "A."

This includes Emacs. To put this to use, let's assign the **`C-t`** keystroke to execute the `kill-word` command, which deletes text from the cursor's position to the next word ending. (This shows my age—I got used to this command when Wordstar was the most popular PC word processor and "32-bit" meant a refrigerator-sized minicomputer. **`M-d`** is the built-in Emacs keystroke that kills the next word.)

Add the following line to your `.emacs` file to make this command available:

```
(global-set-key "^T" 'kill-word)    ; enter ^T here with C-q C-t
```

Setting Emacs Variables to Control Its Behavior

Another common reason to edit your `.emacs` file is to change the value of variables. Programs use variables to keep track of values, and you can change the behavior of your copy of Emacs by changing the value of its built-in variables.

Some variables keep track of numeric values. For example, `next-screen-context-lines` keeps track of how many lines of a given screen remain when you press a **`Page Up`** or **`Page Down`** key. When it's set to 2, the bottom two lines become the top two lines when you page down, and the top two lines become the bottom two when you page up. To check a variable's value, enter **`C-h v`** and then the variable's name.

You can change the value of such variables by entering **`M-x`** and then entering `set-variable` at the Emacs command line. Press **`Enter`**, and Emacs prompts you for the name of the variable to set. After entering its name and pressing **`Enter`** again, Emacs prompts you for the variable's new value.

This is a lot of steps, especially for a variable with a name as long as `next-screen-context-lines`. If you want to set this value to 1 every time you start up Emacs, use the Emacs LISP `setq` function to automate this by adding the following line to your `.emacs` file:

```
(setq next-screen-context-lines 1)
```

Not all variables hold numeric values; some, like `load-path`, hold strings of characters. `load-path` is a list of subdirectories in which Emacs looks when you tell it to load a file of LISP code. As we'll see in the section "Editing SGML Documents with Emacs and PSGML," you'll need to reset `load-path` for Emacs to find PSGML.

Many Emacs variables are Boolean. A Boolean variable (named for the 19th-century British mathematician George Boole) is like a switch that can be turned on or off, and while beginning programmers rarely use them in their programs, the ability to set Emacs's many Boolean variables gives you a great deal of freedom in customizing its behavior. For example, your copy of Emacs may or may not do case-sensitive searches when you execute an incremental search with **`C-s`** or **`C-r`**. (A case-insensitive search finds the character strings "HELLO" or "hello" when you tell it to search for "Hello"; a case-sensitive search only finds an exact match.)

You can control case sensitivity by setting the `case-fold-search` variable. To tell Emacs to ignore case when searching, add the following line to your `.emacs` file:

```
(setq case-fold-search t)
```

To set your default to case-sensitive searching, set the variable off with this line:

```
(setq case-fold-search nil)
```

Other programming languages use "true" and "false" or 1 and 0 to turn Boolean variables on and off. Emacs uses `nil` to turn Boolean variables off for reasons that have more to do with LISP than with Emacs; the `t` is just a convention, because anything other than `nil` assigns a true value. (This is similar to the C-related languages' use of `0` to represent "false" and any other number to represent "true.") Remember this when reading Emacs documentation, which rarely tells you to set a variable to "t"—it's more (mathematically) proper because it tells you to set a Boolean variable to `nil` for one kind of behavior or to a "non-nil" value otherwise. Use `t` for this.

To store a keyboard macro definition in your `.emacs` file so that you can use that macro without ever needing to redefine it, the `insert-kbd-macro` command adds the Emacs LISP equivalent of the macro definition to the current buffer. Let's look at an example.

A handy macro for SGML people is one that speeds the entry of comments. To define a macro, first enter **`C-x (`** to tell Emacs to start recording a macro, and then enter the characters

```
<!--  -->
```

followed by four cursor-left keystrokes, which put the cursor where you can start typing the comment. Next, enter **`C-x )`** to tell Emacs to stop recording.

Name your macro by pressing **`M-x`** and entering `name-last-kbd-macro` at the prompt in the minibuffer. A good name would be `sgml-comment`.

Next, edit your `.emacs` file, or its equivalent in the operating system you're using, and move your cursor to a blank line where you want the macro definition. Press **`M-x`** to bring up the minibuffer prompt and enter the `insert-kbd-macro` command. When Emacs asks which macro to insert, type `sgml-comment`. You'll see the following appear at the cursor:

```
(fset 'sgml-comment [?< ?! ?- ?- ?  ?  ?- ?- ?> left left left left])
```

Now you have a new command in your Emacs environment called “sgml-comment.” Since typing this at the command line is no easier than typing out “<!-- -->” every time you want to enter a comment, you'll want to assign this new command to a keystroke, so add the line

```
(global-set-key “^Co” 'sgml-comment)
```

to assign it to the **C-c o** keystroke. (I chose this because “c” and “o” are the first two letters in the word “comment” and because this keystroke combination is not bound to any existing Emacs function. Remember to enter **C-q** before entering **C-c** so that the **C-c** character is inserted into your `.emacs` file. Enter the “o” as you would normally.) Make sure to save your `.emacs` file. Quit out of Emacs, start it up again with a test file, and enter **C-c o** to see your new saved macro in action.

Available On-line Help

Emacs offers extensive on-line help, all available by pressing the **C-h** key. This displays the following prompt in the minibuffer:

```
C-h (Type ? for further
options)-
```

Emacs offers so many categories of help that you need to tell it which category you're interested in seeing. If you don't know the categories, enter a question mark and Emacs splits the screen, if necessary, to show you descriptions of the various kinds of available help (see Fig. 2.9).

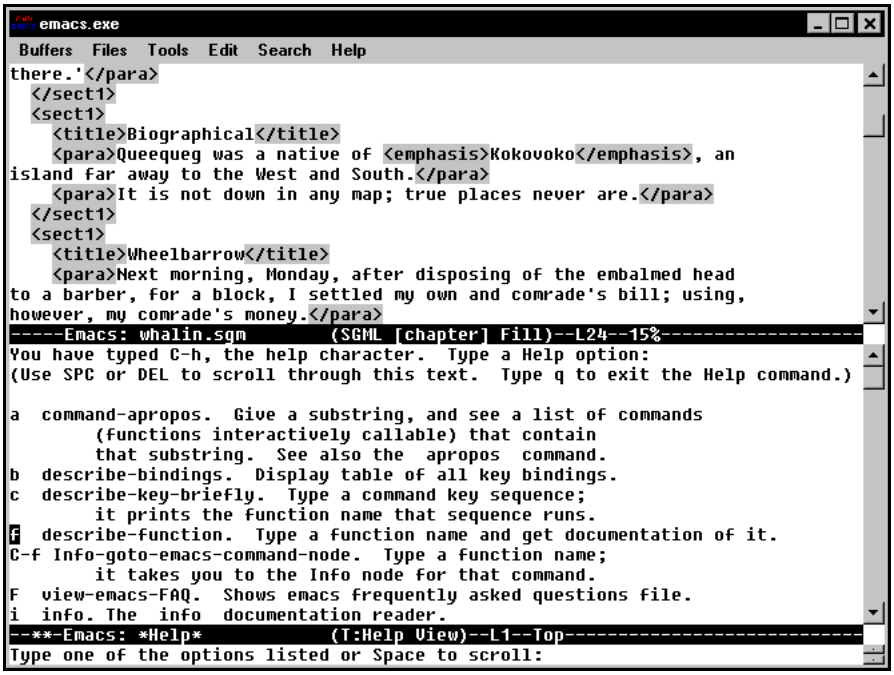

Figure 2.9. Emacs's description of various help categories

There are so many help categories that they won't fit on the screen. Instead of trying to memorize them all, start with the most important ones, and remember that you can always press **C-h ?** to learn more.

a (`command-apropos`)

List all the commands with a given string of characters in them. For example, enter **C-h a** and then the word "macro" in response to the `Command apropos (regexp):` prompt in the minibuffer. Fig. 2.10 shows how the help facility lists all the commands that involve creating and using macros (with their keystrokes if any have been assigned).

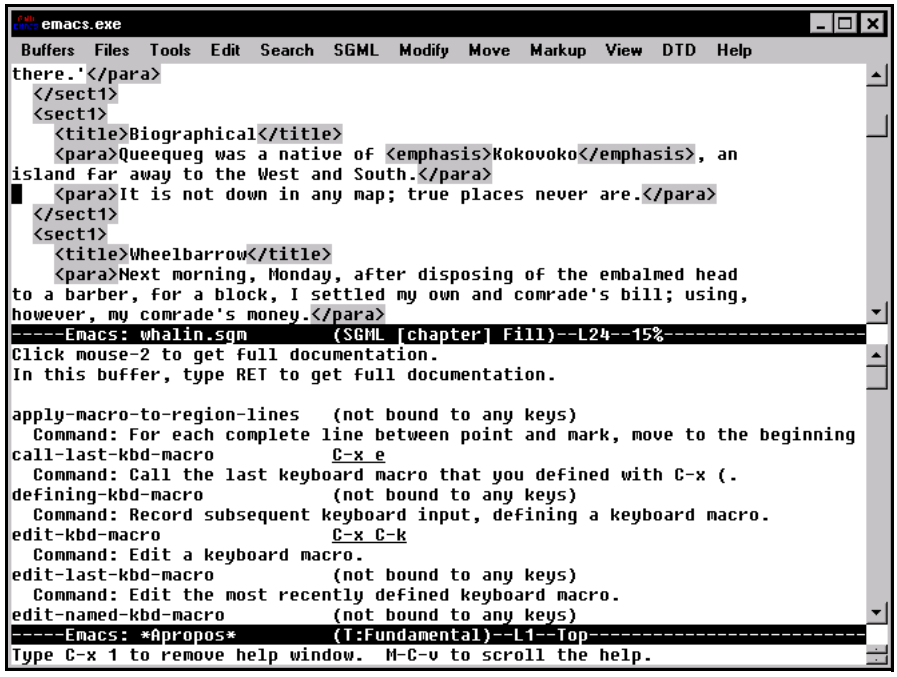

Figure 2.10. `command-apropos` entries for "macro"

Selecting **Command Apropos...** from the **Help** menu also invokes this command.

`C-h k` (`describe-key`)

After you press **`C-h k`**, press any Emacs keystroke to display on-line help about that keystroke. This is useful for deciding which keystroke is free to have a macro assigned to it. For example, pressing **`C-h k C-a`** displays the screen shown in Fig. 2.11.

Selecting **Describe Key...** from the **Help** menu does the same thing.

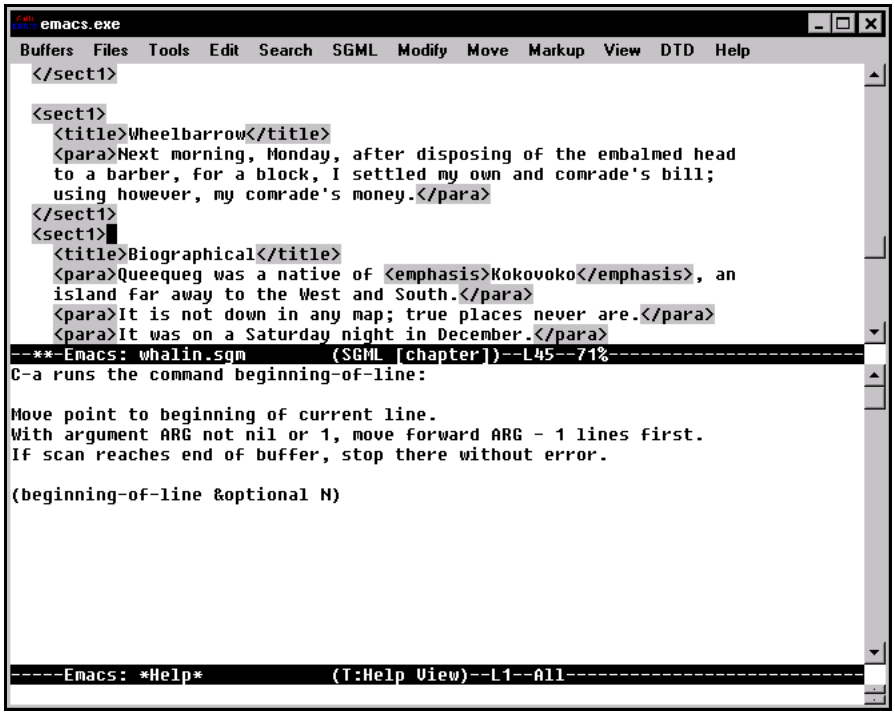

Figure 2.11. `describe-key` information on **`C-a`**

`C-h t` (`help-with-tutorial`)

Start the interactive Emacs tutorial. This reviews the Emacs features that you read about here and teaches you some new ones.

`C-h v` (`describe-variable`)

Enter the name of an Emacs variable and the help system displays the variable's current value and any available documentation. This is useful to check on the necessity of resetting a variable's value.

C-h i (info)

Start the `info` program, a menu-driven browser for Emacs documentation, as shown in Fig. 2.12.

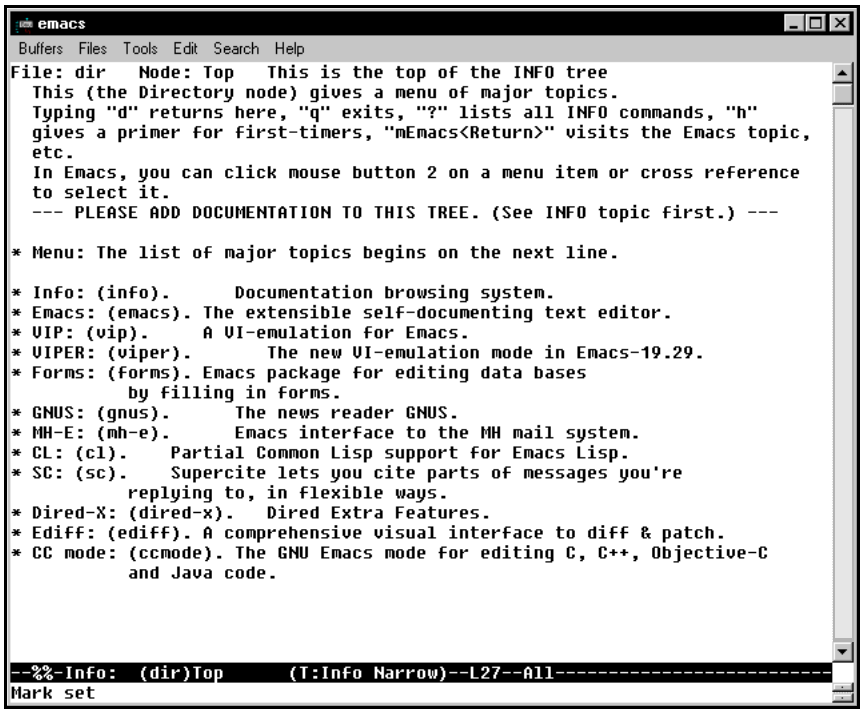

Figure 2.12. Emacs `info` opening screen

The lines with asterisks are menu choices. Move your cursor to one and press **Enter** to select that choice. The "Info" choice is a good place to start in order to learn more about navigating the Info browser. The handiest Info command is **?**, which displays the important single-letter Info navigation commands. The most important of these is the letter `l` for "last node," which lets you retrace your path after selecting a series of Info nodes (information topics).

If your copy of Emacs has trouble finding the Info files, add the following line to your `.emacs` file to tell Emacs where to look, substituting the appropriate pathname for `/pathname/emacs/`:

```
(setq
Info-directory-list      (cons
"/pathname/emacs/info" Info-default-directory-list))
```

Entering a question mark after you've entered **`C-h`** is like asking for help about the help facility. After you've looked it over, you're still in the on-line help, and Emacs is still waiting for you to enter one of the other possible keystrokes (besides **`?`**) that can come after **`C-h`**— such as **`a`** for `command-apropos` or **`i`** for `info`.

Editing SGML Documents with Emacs and PSGML

Emacs actually has a built-in mode for editing SGML documents, but it merely automates the insertion of tag delimiters ("<" and ">") and the calling of the external program that can validate the file you're working on, such as nsgmls or sgmls.

PSGML is an add-on SGML mode for Emacs which does much more. While not a complete SGML validator, it adds many features to Emacs that let you edit a document instance without worrying about its structure because it automates most of the tasks necessary to maintain that structure. In addition to letting you select and add the appropriate tags with a minimum of keystrokes, it can insert required tags automatically, help you find structural mistakes, indent tags to show nesting structure, and let you set tags, comments, and entity references in different fonts or colors to make it easier to see the structure.

Installing PSGML

A user usually receives PSGML as a collection of files combined into one compressed file such as a DOS/Windows ZIP file or a UNIX GNU zip file. Files extracted from this distribution file fall into three main categories:

- Those that end with an extension of `.el`, which are collections of LISP functions similar to your `.emacs` file.
- Those with an extension of `.elc`, which are compiled versions of the `.el` file that run faster.
- All the other files, which contain information about installing and using PSGML.

UNIX users will find installation instructions and scripts included with PSGML. DOS/Windows users need to execute the important steps by hand.

The `.elc` files created with one version of Emacs on a particular operating system may not work with another version of the program or on another operating system, but you can create new ones yourself from the `.el` files. You can actually run PSGML using only the `.el` files, but it's so much faster with `.elc` files that you may as well create them before trying to use it.

First, put all the PSGML files into their own subdirectory and make sure that Emacs knows where to find them by adding the following LISP code to your `.emacs` startup file:

```
(setq load-path
 (append
       (list nil
                 "/app/emacs/psgml")  ; substitute your PSGML directory name
 load-path))
```

<TIP> **Even when using Emacs and PSGML under DOS/Windows, PSGML expects the directory name to be entered with forward slashes ("/") and not the backslashes ("\") used in PC pathnames.**

This adds the directory name to the Emacs `load-path` variable, a list of directories where Emacs looks when you tell it to load a particular program. Substitute the name of the directory where you stored the PSGML files for `/app/emacs/psgml` above.

Next, enter **`M-x`** to display the Emacs command line and enter `byte-compile-file`. Press **`Enter`**, and at the `Byte compile file:` prompt, enter the name of an `.el` file to compile and press `Enter`. If the prompt is `Byte compile file: ~/`, you can treat that `~/` as the name of the current directory and then type `psgml/psgml.el` after it if you want to compile the `psgml.el` file in the `psgml` subdirectory of your current directory.

Do this for all the `.el` files in your PSGML directory. If you see any error messages during the compilation of a file, compile the others and go back and try that one again. It may have relied on something from another `.elc` file that you hadn't yet created.

A shortcut to repeatedly entering the `byte-compile-file` command is the `byte-force-recompile` command, which prompts you for the name of a directory and then compiles all the `.el` files in that directory.

I mentioned that Emacs has a built-in SGML mode that doesn't do much, so the next step is to tell Emacs to use PSGML for its SGML mode instead of the built-in one. Do this by adding the line

```
(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

to your `.emacs` file.

In addition to the program files, PSGML includes documentation that you can view with the Emacs Info browser. Put the `psgml.info` file into the directory (probably a subdirectory of the main Emacs directory called `info`) with the other Info files. Then edit the `dir` file in that same directory to include the following line:

```
* PSGML: (psgml).       SGML editing.
```

If it's a multi-user system and you're not the system administrator, you probably won't be allowed to add or edit files in a subdirectory of the Emacs directory. You can still view the `psgml.info` file from within Emacs by putting it in any directory you wish and entering the command

```
C-h i g (/pathname/psgml.info)
```

to display that file with the Info program. (Include the full pathname of its location where you see `/pathname/`.)

<TIP> **Remember to leave off the "o" in the file extension if you're using Emacs and PSGML under DOS, where file extensions can only be up to three letters long.**

When specifying the info file's location, DOS/Windows Emacs accepts pathnames with forward slashes ("/") or backslashes ("\") separating its components.

Starting Up PSGML

How does Emacs know to load PSGML mode? There are three ways:

- It knows to automatically load it if you edit a file with an extension of ".sgml" or ".sgm."

- You can load a mode in Emacs by entering **M-x** and entering the mode's name (in this case, `sgml-mode`) at the minibuffer command line. This is valuable when editing SGML files that don't have ".sgml" or ".sgm" as an extension.
- The following line at the top of your file tells Emacs to automatically load PSGML:

```
<!-- -*- sgml -*- -->
```

Once you start up Emacs with PSGML and load a document, PSGML doesn't actually parse the document (unless the `sgml-auto-activate-dtd` and `sgml-set-face` variables are both set to `t`) until you execute one of a certain category of commands. Because of this, it can't do any of the automatic visual formatting that makes editing your document easier—for example, indenting of the element nesting levels or displaying markup in different fonts or colors. One of the commands that causes it to parse, and a good one to start with, is the `sgml-next-trouble-spot` command, which you invoke by pressing **C-c C-o** or by selecting **Next Trouble Spot** from the **Move** menu. While PSGML is not a full validating parser (and it can call one easily enough, as we'll see in "Finding Tagging Mistakes") it can locate many potential problems, and the parsing that it performs while looking lets it do this visual formatting.

The most common problem that `sgml-next-trouble-spot` finds is a tag that doesn't belong somewhere, either because it's in the wrong place or because it's not even a valid tag for that document (for example, if the generic identifier has a typo). If it finds no problems, the cursor jumps to the end of the document, finishes up any visual formatting, and displays the message `Ok` in the minibuffer.

With a large DTD, this parsing can take a while. You'll see messages like "Parsing doctype" and "Garbage collecting" (a colorful term for "RAM reorganization") flash by for a while in the minibuffer. To speed this process, you can save the DTD in

a special compiled version that PSGML loads more quickly by entering the command `sgml-save-dtd` at the **`M-x`** minibuffer prompt. PSGML offers to save the compiled version in the same directory with the same filename as the document you are editing but with an extension of `.ced`. If you do change the default directory or filename at all, you'll have to specify the full pathname of the saved one when you want to use it with the command `sgml-load-dtd`. If you let PSGML save the compiled one with the default name and directory it can find this compiled version by itself the next time you edit that document.

SGML Declarations and DTDs

PSGML doesn't require you to include an SGML declaration with your document. In fact, if you do include it, PSGML ignores it. It uses the Reference Concrete Syntax but ignores the Reference Concrete Syntax's limitation on element type name length.

PSGML must know how to find your document's DTD so that it can enforce your document instance's structure. There are three ways to tell PSGML how to locate the DTD to use with your document:

- A SYSTEM identifier in the DOCTYPE declaration.
- A PUBLIC identifier in the DOCTYPE declaration.
- One of several Emacs variables set by PSGML.

The first two methods are standardized and are therefore used by other SGML applications. For this reason, the beginner is best off ignoring the third method.

The SYSTEM identifier is the simplest method. Within a DOCTYPE definition, such as the following,

```
<!DOCTYPE chapter SYSTEM "mybook.dtd">
```

the keyword SYSTEM tells the software "the DTD is in the following file on this system." Using the above declaration, on most systems the software looks in the document file's directory. With some systems you can also specify a relative or absolute pathname with the file; the following shows a DTD filename with an absolute filename:

```
<!DOCTYPE chapter system "\dev\sgml\dtds\mybook.dtd">
```

Some SGML software is pickier about the pathname, expecting UNIX-style forward slashes ("/") instead of backslashes ("\"), even if running a DOS or Windows system that uses backslashes to identify pathname components. PSGML has no problem with the DOS style.

The PUBLIC identifier is popular when using well-known DTDs. The DOCTYPE declaration includes a string after the word PUBLIC, such as this one for the DocBook DTD:

```
<!DOCTYPE chapter PUBLIC "-//Davenport//DTD DocBook V3.0//EN">
```

The SGML application software (in this case, PSGML) still needs to know where a copy of this public DTD is, so the most common way to tell it is with a catalog file following the format defined by SGML Open Technical Resolution 9401:1995. Each entry of this file (commonly called `catalog`) has the keyword PUBLIC followed by the string that identifies the DTD and the actual filename (and, if necessary, the location) of the system's copy of the DTD. The following shows typical catalog entries for the DocBook and HTML DTDs:

```
PUBLIC "-//Davenport//DTD DocBook V3.0//EN"
 "DOCBOOK.DTD"
PUBLIC "-//IETF//DTD HTML//EN"
 "\WEBSTUFF\HTML.DTD"
```

Note that the DocBook entry has no pathname, indicating that the `docbook.dtd` file is in the same directory as the `catalog` file, while the HTML entry includes an absolute pathname.

Once the SGML application software knows where the `catalog` file is, it can look there to see which DTD file to use with the DOCTYPE's public declaration. To tell PSGML where to find the `catalog` file, set the environment variable `SGML_CATALOG_FILES` to the file's fully qualified name (that is, the name with its pathname included). If the file is called `catalog` and stored in a directory named `\dev\sgml\dtds\` on a DOS/Windows system, the following line tells PSGML where to find the catalog:

```
set SGML_CATALOG_FILES=\dev\sgml\dtds\catalog
```

<TIP> **The UNIX syntax for setting the environment variable depends on the shell you're using.**

Keep in mind that a DTD itself may contain declarations for public entity sets that it uses, and that you may need to tell the application software where to find those entity sets. For example, if your DTD includes the following entity declaration to let documents use the public ISO character entity set,

```
<!ENTITY % ISOchars PUBLIC
                "-//ENTITIES Public ISO Character Entity Declarations//EN"
 "iso-public">
```

then your catalog file needs an entry to indicate that character entity set's location. This means adding an entry like this to your catalog file:

```
PUBLIC "-//ENTITIES Public ISO Character Entity Declarations//EN"  "ISOPUB.ENT"
```

Using PSGML

PSGML makes it easier to edit SGML files by adding commands to manipulate SGML document instances and by adding keystrokes and menu choices that invoke most of these commands. Even if the version of Emacs and the operating system that you use cannot display fonts, colored text, or menus, PSGML's built-in intelligence still offers help to the writer creating a document. It reads and understands your DTD and automates the chores of ensuring that your document conforms to it, letting you concentrate on the content you're creating and not on the correctness of the tags that describe your document's structure.

PSGML Startup Variables

In the section "Setting Emacs Variables to Control Its Behavior," we saw that Emacs has variables that you can set in the `.emacs` file to customize its behavior. PSGML adds several new ones, and I've found the following Boolean ones most valuable when I use PSGML. I set all of them to `t` in my `.emacs` file with a line like this:

```
(setq sgml-omittag-transparent t)
```

sgml-omittag-transparent

When you tell PSGML that you want to insert a new element, you can have it list all the allowable element types that you may insert at the cursor's current position. If `sgml-omittag-transparent` is set to a nil value, PSGML only lists element types that can occur inside the element where the cursor is located. If set to a non-nil value, PSGML also lists element types that can begin after what PSGML calls the "live" element (that is, the element where the cursor is

currently located) if the live element doesn't need an end-tag.

For example, if the live element is a `para` element and the DTD doesn't require `para` elements to have an end-tag, any commands that list valid element types to insert at the cursor position will list those that may come after a `para` element as well as those that may appear within one.

sgml-balanced-tag-edit

If this is set to a non-nil value, each time you tell PSGML to insert a certain element, it inserts the beginning and end-tag for you, putting your cursor between them.

sgml-auto-insert-required-elements

If this is set to a non-nil value, then each time you tell PSGML to insert a particular element, it also inserts the tags for any required elements within that element. For example, when using the DocBook DTD, telling PSGML to insert the `chapter` element also inserts the tags for the `title` element at the beginning of the chapter and follows it with a comment telling you to insert "one of (list of valid element types)" to tell you that the title must be followed by at least one of the listed element types.

sgml-set-face

If set to a non-nil value, this variable tells PSGML to display (if possible on your monitor) tags, content, entity references, and comments in different fonts or colors. For more on this, see the section "Displaying Markup in Different Fonts and Colors."

sgml-live-element-indicator

Setting this to a non-nil value tells PSGML to display the document type and the live element type on the mode line. For example, if your cursor is on a `title` element while editing a DocBook `chapter`, the mode line might have the following message:

```
(SGML [chapter/title]))
```

This may slow down Emacs a bit, but the sacrifice is often worth it if your document isn't too large.

sgml-indent-step

This sets PSGML to automatically indent tags as you type to give visual clues about element structure. With the `sgml-indent-step` variable set to its default value of two, the tags of an element inside another are indented two more spaces from the left than the tags of its container element. In the following, we see that the `title`, `para`, and `figure` elements' tags are all indented two more characters than the `sect2` start- and end-tags, and that the `figure` element's `title` and `graphic` subelements are indented two more characters than the `figure` start- and end-tags:

```
<sect2>
  <title>A Bosom Friend</title>
  <para>We then turned over the book together, and I endeavored to
explain to him the purpose of the printing, and the meaning of the few
pictures that were in it.</para>
  <figure>
    <title>A Sample Figure</title>
    <graphic fileref="giftest.gif" format="gif"></graphic>
  </figure>
  <para>Thus I soon engaged his interest; and from that we went to
```

```
jabbering the best we could about the various outer sights to be seen
in this famous town.</para>
<sect2>
```

Too much indenting can push text too far to the right side of your screen, so you can turn indenting off by setting `sgml-indent-step` to a value of zero in your `.emacs` file with the following line:

```
(setq sgml-indent-step 0)  ; default value is 2
```

Entering PSGML Commands

As with other Emacs commands, all PSGML commands can be entered at the **`M-x`** command prompt, most can be entered with keystrokes, and many can be entered through menus if your version of Emacs supports menus.

Keystrokes are usually the fastest. Most PSGML keystrokes consist of **`C-c`** followed by another Control key combination; many are the SGML equivalents of regular Emacs commands. For example, in regular Emacs, **`M-f`** moves the cursor one word forward and **`M-b`** moves the cursor one word backward. In "Moving Your Cursor Around an SGML Instance," we'll see that **`C-M-f`** and **`C-M-b`** move the cursor forward and backward one element. (The **`C-M-f`** notation can be a bit confusing; think of it as **`Escape`** followed by **`Ctrl+f`**, not **`Escape`** between the **`Ctrl`** and **`f`**, as it's written. Some versions of Emacs let you press **`Alt`** instead of **`Escape`** for metakeys, so that **`C-M-f`** could be **`Ctrl+Alt+f`**.)

PSGML adds six new menus to the regular Emacs selection: **SGML**, **Modify**, **Move, Markup**, **View**, and **DTD**. As this section covers each PSGML feature, it mentions any menu alternatives to commands and keystrokes.

Entering Text

The `sgml-insert-element` command could be the one you use the most when writing with PSGML. When you enter it at the command line (or when you select **Insert Element** from the **Markup** menu, or more likely, when you enter the **`C-c C-e`** keystroke that invokes it), PSGML prompts you for the name of the element type in the minibuffer:

```
Element:
```

As with other minibuffer prompts, you can use Emacs's completion feature, so you only need to type the first few characters at this prompt and then press **`Tab`** or the space bar to have Emacs fill out as much as it can. If more than one valid element type name begins with the letters you typed (and note that "valid" part—the fact that this command only offers you valid element type names to insert makes it very useful), Emacs lists them in a new window so that you can choose one.

After you tell PSGML the element type to insert, if `sgml-balanced-tag-edit` and `sgml-auto-insert-required-elements` are set to non-nil values (see "PSGML Startup Variables" for more on these) then PSGML inserts the element's start-tag, end-tag, and any required subelements. If a certain point in the document needs an element from a choice of several types, PSGML inserts an SGML comment that lists your choices. Finally, it puts your cursor right after your new element's start-tag (or the first subelement in it that requires text, as we'll see in the `figure` example below) so that you can start typing.

For example, let's say you want to insert a figure into a DocBook document. First press **`C-c C-e`**, which displays the prompt `Element:` in the minibuffer to ask what element type to insert. Enter the letters `fi` and press **`Tab`** to see if Emacs enters the remaining letters of the word "figure" for you. No other DocBook element types begin with the letters "fi," so

Emacs doesn't need to open a separate window to ask “which element type begins with 'fi'?” and it adds the “gure” for you in the minibuffer.

When you press **Enter**, PSGML inserts the following text and positions your cursor between the `title` element's start- and end-tags:

```
    <figure>
      <title></title>
      <!-- one of (blockquote informalequation informaltable literallayout pr\
ogramlisting screen screenshot graphic synopsis cmdsynopsis funcsynopsis link o\
link ulink) -->
    </figure>
```

Note how the line after the `title` tags is a comment telling you that you must include one of the listed element types there. The comment is too long to fit on a single line, so it wraps onto a second and third line on your screen and puts a backslash (\) at each wrapping point to show you that the text after each slash is a continuation of the current line. Although the line is visually wrapped so that you can see the whole thing, the backslashes show that it's still technically one line, which makes it easier to delete it once you've heeded the comment's advice: simply put your cursor at the beginning of the comment and press **C-k** to invoke the `kill-line` command.

This description might make the process of inserting a `figure` element appear complicated, but let's review it without the background information to see how little work there really is:

1. Press **C-c C-e**.
2. Enter `fi` at the `Element:` prompt and press **Tab**.
3. Press **Enter** and start typing the text of the figure's title.

Sometimes you can't start typing right away because the content model of your new element offers a choice of subelements at the point where you start entering text. For example, you can start a DocBook `ItemizedList` (such as a bulleted list) by

entering **C-c C-e**, the letters "it," and pressing **Tab** and **Enter** to let Emacs enter the rest of the element type name. PSGML enters the tags to start the list and a `ListItem` start-tag for the list's first item, but the `ListItem` can begin with so many different element types that PSGML inserts a 400-character comment to tell you about your choices:

```
    <ItemizedList>
      <ListItem>
 <!-- one of (sidebar procedure msgset simpara para formalpara funcsynop\
sis cmdsynopsis synopsis graphic screenshot screen programlisting literallayout\
 informaltable informalequation blockquote variablelist simplelist segmentedlis\
t orderedlist ItemizedList indexterm highlights table figure example equation e\
pigraph comment bridgehead warning tip note important caution authorblurb ancho\
r abstract) -->
      </ListItem>
    </ItemizedList>
```

If you're entering simple text, you'll probably want to enter a `para` ("paragraph") element.

If `para` isn't the only valid element type beginning with the letter "p," you'll probably have to enter at least "pa" at the `Element:` minibuffer prompt before you press **Tab** to tell Emacs to fill out the rest of the element type name for you. Since the element type representing a simple paragraph of text is the one you'll probably enter most often into any document, it's good to know that the PSGML `sgml-split-element` keystroke lets you do this with even fewer keystrokes than the **C-c C-e** method.

Splitting the Live Element

The `sgml-split-element` command, invoked with **C-c Enter**, tells PSGML to split the live element into two at the cursor position. If you decide that an existing paragraph should actually be two paragraphs, pressing this keystroke with your cursor in that paragraph puts an end-tag and start-tag at the cursor location and positions your cursor right after the

new start-tag. For example, if you want the sentence beginning "Go from Corlears Hook" to begin a new `para` element in Fig. 2.13,

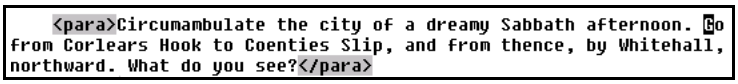

Figure 2.13. Using `sgml-split-element` to split an existing element

pressing **`C-c Enter`** with your cursor on the "G" in "Go" has the result shown in Fig. 2.14.

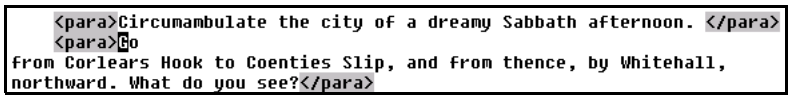

Figure 2.14. Result of splitting an element with `sgml-split-element`

You might use the Emacs **`M-q`** `fill-paragraph` keystroke to adjust those line breaks, but we'll see a better way that takes element structure into account in the "Justifying Element Text" section.

With your cursor just before a paragraph end-tag, **`C-c Enter`** essentially ends the current paragraph and starts a new empty one. This is great when entering new text (as opposed to editing existing text) because it enters a large majority of your first draft tags for you. For example, let's say your cursor is right after the question mark in Fig. 2.15.

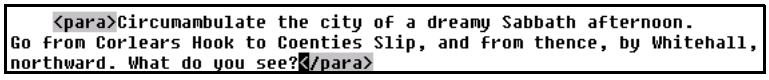

Figure 2.15. Creating a new element with `sgml-split-element`

Press **C-c Enter**, and PSGML starts a new paragraph after that one and puts your cursor between the new paragraph's start- and end-tags, so that you can type away without having entered any tags, element type names, or other structural information (see Fig. 2.16).

```
    <para>Circumambulate the city of a dreamy Sabbath afternoon.
Go from Corlears Hook to Coenties Slip, and from thence, by Whitehall,
northward. What do you see?</para>
    <para></para>
```

Figure 2.16. Result of creating a new element with `sgml-split-element`

So, unless you're writing highly specialized text, over three-quarters of the process of writing with Emacs and PSGML consists of entering your text and pressing **C-c Enter** every few sentences. The keystroke is easy to remember because we already saw that most PSGML keystrokes consist of **C-c** followed by other characters, and this particular key is the simplest to follow it—in fact, the **Enter** key is what you'd press at the end of a paragraph with nearly any word processor.

Keep in mind that the `sgml-split-element` keystroke does more than just create new paragraphs. One nice trick is pressing **C-c Enter** a second time to split the live element's parent, or pressing it a third time to split the parent's parent. For example, let's say you just finished typing "may be." in Fig. 2.17.

```
  <sect1>
    <title>Gone Whalin'</title>
    <sect2>
      <title>Loomings</title>
      <para>Let us scrape the ice from our frosted feet, and see what
sort of a place this 'Spouter' may be.</para>
    </sect2>
  </sect1>
```

Figure 2.17. Before splitting any elements

Pressing **C-c Enter**, as we already saw in this situation, starts a new paragraph under the current one, as shown in Fig. 2.18.

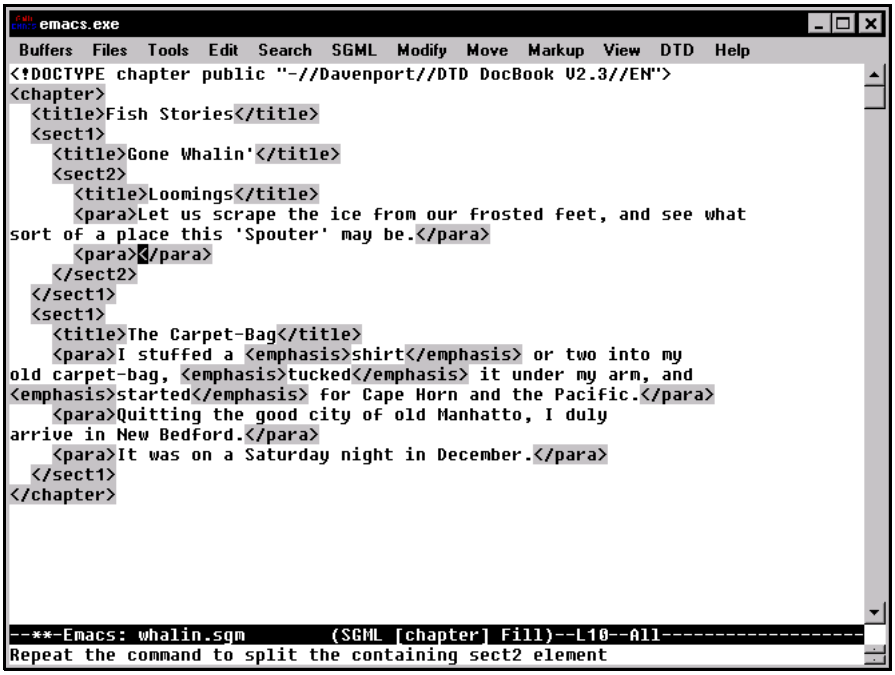

Figure 2.18. Splitting the `para` element

Note that after doing this, a minibuffer message tells you `Repeat the command to split the containing sect2 element.` In other words, because that `para` element is inside a `sect2` element, immediately pressing **C-c Enter** again ends the current `sect2` element and starts a new one, putting your cursor between the `para` start- and end-tags that begin our new `sect2` element (see Fig. 2.19).

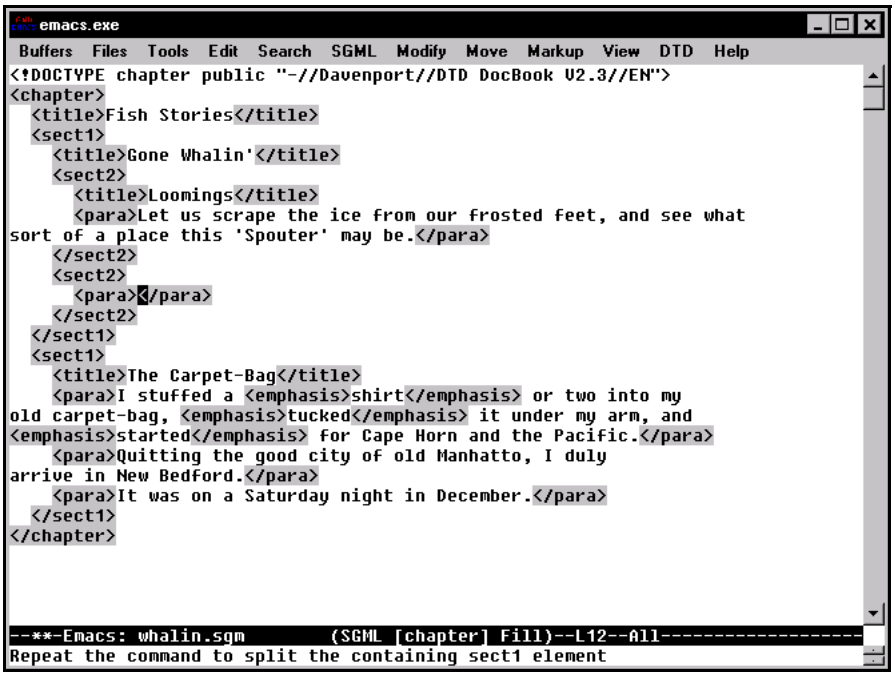

Figure 2.19. Splitting the `sect2` element by repeating the **`C-c Enter`** keystroke

Remember that this doesn't automatically insert other elements that may be necessary—in this case, the new `sect2` element lacks its required `title` element—so you'll have to remember the occasional `sgml-next-trouble-spot` keystroke (**`C-c C-o`**) to check for such oversights.

Now the minibuffer message tells you `Repeat the command to split the containing sect1 element`. So, pressing **`C-c Enter`** one more time ends the `sect1` element titled "Gone Whalin'" and starts a new `sect1`, putting a new `sect2` element inside of the new `sect1` and a new `para` inside of the new `sect2` (see Fig. 2.20).

```
  <sect1>
    <title>Gone Whalin'</title>
    <sect2>
      <title>Loomings</title>
      <para>Let us scrape the ice from our frosted feet, and see what
sort of a place this 'Spouter' may be.</para>
    </sect2>
  </sect1>
  <sect1>
    <sect2>
      <para></para>
    </sect2>
  </sect1>
  <sect1>
```

Figure 2.20. Splitting the `sect2` element's enclosing `sect1`

Because the new `sect1` needs a title, `sgml-next-trouble-spot` will give you similar error messages about starting a new paragraph without including the `sect1` element's necessary `title` element.

For a simpler, more typical example, let's say you're finishing a paragraph within a list's first item and you want to begin a new list item (see Fig. 2.21).

```
    <itemizedlist>
      <listitem>
        <para>But what thinks Lazarus? Can he warm his blue hands by
        holding them up to the grand northern lights?</para>
      </listitem>
    </itemizedlist>
```

Figure 2.21. Ready to add a new list item

After typing the question mark following the word "lights," press **`C-c Enter`** and PSGML puts a new paragraph in that list item, as shown in Fig. 2.22.

```
<itemizedlist>
  <listitem>
    <para>But what thinks Lazarus? Can he warm his blue hands by
    holding them up to the grand northern lights?</para>
    <para></para>
  </listitem>
</itemizedlist>
```

Figure 2.22. Splitting the `para` element with **`C-c Enter`**

What you really wanted, however, was to start a new list item, not a new paragraph within the current list item. Pressing **`C-c Enter`** another time creates this new list item because `listitem` is the parent of the `para` element that you just split. It even puts the start- and end-tags for a new `para` element in that list item, with your cursor between these two tags so that you can start typing right away (see Fig. 2.23).

```
<itemizedlist>
  <listitem>
    <para>But what thinks Lazarus? Can he warm his blue hands by
    holding them up to the grand northern lights?</para>
  </listitem>
  <listitem>
    <para></para>
  </listitem>
</itemizedlist>
```

Figure 2.23. Splitting the `listitem` element by pressing **`C-c Enter`** a second time

To review these steps without all the commentary (and to show how few steps it really was): you finished a list item, pressed **`C-c Enter`** twice, and were ready to start typing the text of the new list item.

Inserting Tags Around Existing Text

So far, inserting elements has meant telling PSGML to insert a start- and end-tag pair and to position the cursor between them. This is convenient when entering new text into a document, but what about adding tags to existing text? For example, when putting a pair of `emphasis` tags around a word in the

middle of a sentence? Using the `sgml-tag-region`, `sgml-insert-tag` and `sgml-insert-end-tag` commands and keystrokes (along with the convenience of Emacs's completion feature) makes this simple.

The `sgml-tag-region` keystroke (**`C-c C-r`**) and the **Tag Region** choice of the **Markup** menu tell PSGML to put a start-tag at the beginning of the region and the corresponding end-tag at the end. (Remember, the "region" is like a marked block in other word processors and text editors, delimited by your last marked point and the cursor's current location.) For example, let's say you put your cursor on the "R" in "Right" in Fig. 2.24, pressed **`C-space`** to set the mark there, and moved your cursor to the comma after "Whalemen."

```
      <para>South-eastward from the Cape, off the distant Crozetts, a
good cruising ground for Right Whalemen, a sail loomed ahead, the
Goney (Albatross) by name.</para>
```

Figure 2.24. Just before adding `emphasis` tags

Pressing **`C-c C-r`** displays the prompt `Tag region with element:` in the minibuffer, waiting for you to enter an element type name. You can use completion, so to put `emphasis` tags around the phrase (when using the DocBook DTD) you only need to type "em" followed by a **`Tab`** to have PSGML fill in the "phasis" and then **`Enter`** to complete the command, as shown in Fig. 2.25.

```
      <para>South-eastward from the Cape, off the distant Crozetts, a
good cruising ground for <emphasis>Right Whalemen</emphasis>, a sail loomed
ead, the
Goney (Albatross) by name.</para>
```

Figure 2.25. `emphasis` tags added with **`C-c C-r`**

(The paragraph may look like it needs the `fill-paragraph` keystroke (**`M-q`**), but we'll see an even better PSGML version of this command in the "Justifying Element Text" section.) Some versions of Emacs don't highlight the marked region, so it's

easy to forget where you set the last mark, and sometimes the `sgml-tag-region` command puts the start- or end-tag where you didn't expect it to. To make sure of the mark's location, the `exchange-point-and-mark` keystroke (**`C-x C-x`**) mentioned earlier is a great way to quickly check. Press it twice (in other words, press **`C-x`** four times) to check the mark's location and then return the cursor to its original position.

The `sgml-insert-tag` and `sgml-insert-end-tag` commands let you insert start and end-tags individually. **`C-c <`** (and **Insert Start-Tag** from the **SGML** menu) invoke the `sgml-insert-tag` command, and **`C-c /`** (and **Insert End-Tag** from the same menu) invoke the `sgml-insert-end-tag` command. To put `emphasis` tags around the word "No" in Fig. 2.26, first put your cursor at the "N."

```
      <para>Unconsciously clapping the vinegar-cruet to one side of
her nose, she ruminated for an instant; then exclaimed--'No! I haven't
seen it since I put it there.'</para>
```

Figure 2.26. Before inserting `emphasis` start-tag

Press **`C-<`**. At the `Tag: <` prompt, you only need to enter "em" when using the DocBook DTD and then **`Tab`** to complete the element name. Press **`Enter`** to insert the start-tag as shown in Fig. 2.27.

```
      <para>Unconsciously clapping the vinegar-cruet to one side of
her nose, she ruminated for an instant; then exclaimed--'<emphasis>No! I haven'\
t
seen it since I put it there.'</para>
```

Figure 2.27. `emphasis` start-tag inserted with **`C-<`**

Entering the `emphasis` end-tag requires even less typing because it's the only end-tag that can appear right after an `emphasis` start-tag. With your cursor on the exclamation point, press **`C-c /`**, and that's it—you don't even have to press **`Enter`**. PSGML enters the `emphasis` end-tag for you.

Inserting Elements with the Menu

If your copy of Emacs has a menu, selecting **Insert Element** from the **Markup** menu displays a new menu under the menu bar's **Markup** choice with a list of element types that are valid at the cursor's current position. If only one element type is valid, only one shows up there; if there are too many valid choices to vertically fit on the screen, PSGML replaces the **Markup** menu with a menu offering multiple submenus of the valid element types. For example, with your cursor right after a DocBook `para` element, selecting **Insert Element** from the **Markup** menu displays the element type menus shown in Fig. 2.28.

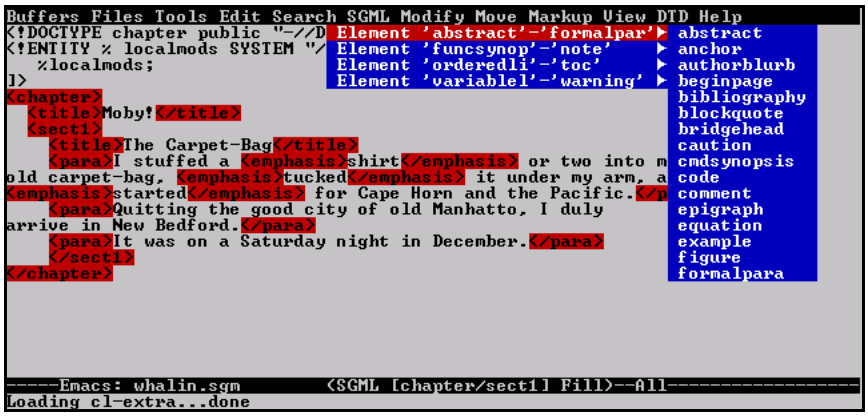

Figure 2.28. First of four menus of DocBook elements that can follow a `para` element

To insert individual start- or end-tags, select **Insert Start-Tag** or **Insert End-Tag** from the same menu.

Justifying Element Text

We saw that **`M-q`** (`fill-paragraph`) justifies a paragraph in Emacs, redistributing words in the paragraph to adjust the line lengths to just under the right margin setting. An SGML docu-

ment instance may have text—whether data, tags, or some combination—on every line, so pressing **M-q** would combine a lot of text into a huge paragraph.

Instead, use the `sgml-fill-element` command, available by pressing **C-c C-q** or by selecting **Fill Element** from the **Modify** menu. In technical terms, it fills the biggest enclosing mixed-content element or fills the subelements if the enclosing element only has element content. In layman's terms, the first part of this tells us that PSGML doesn't justify the text of an element that only has data (such as `emphasis` in the example below). Instead, it justifies the container element in which those elements occur alongside other data.

```
<sect1>
 <title>The Chase--Third Day</title>
    <para>But aye, old mast, we both grow old together; sound in our
hulls, though, are we not, my ship? Aye, minus a leg, that’s all. By
heaven this dead wood has the better of my live flesh every
way.</para>
    <para>I can’t compare with it; and I’ve known some ships made of
dead trees outlast the lives of men made of the most vital stuff of
vital fathers. <emphasis>What’s that he said?</emphasis> he should still go bef\
ore me, my pilot; and yet to be
seen again? But where? Will I have eyes at the bottom of the sea,
supposing I descend those endless stairs and all night I’ve been
sailing from him, wherever he did sink to.</para>
</sect1>
```

In the above example, I just added the emphasized phrase “What's that he said?,” which obviously threw off the paragraph's line breaks. Whether my cursor is on the word “said” between the `emphasis` tag pair or on the word “fathers” to the left of the `emphasis` start-tag, pressing **C-c C-q** justifies the paragraph to look like this:

```
<sect1>
 <title>The Chase--Third Day</title>
    <para>But aye, old mast, we both grow old together; sound in our
hulls, though, are we not, my ship? Aye, minus a leg, that’s all. By
heaven this dead wood has the better of my live flesh every
way.</para>
```

```
    <para>I can't compare with it; and I've known some ships made of
dead trees outlast the lives of men made of the most vital stuff of
vital fathers. <emphasis>What's that he said?</emphasis> he should
still go before me, my pilot; and yet to be seen again? But where?
Will I have eyes at the bottom of the sea, supposing I descend those
endless stairs and all night I've been sailing from him, wherever he
did sink to.</para>
</sect1>
```

If I had pressed the regular Emacs `fill-paragraph` keystroke (**`M-q`**), it would have combined that paragraph with the paragraph before it and any other text after the most recently skipped line and before the next one.

According to the second part of the `sgml-fill-element` command's technical description in the PSGML Info file, if the cursor is in an element that only has element content, with no data other than those in its subelements, then PSGML rejustifies all of that element's subelements. In the last example, if the cursor had been right after the `sect1` start-tag, pressing **`C-c C-q`** would have also justified the paragraph with the `emphasis` element, along with the other `sect1` subelements.

With your cursor at the beginning of a document, this technique is a handy way to clean up an entire document before saving it.

Indenting Text

We saw in the "PSGML Startup Variables" section that the `.emacs` file's `sgml-indent-step` variable makes it possible to have PSGML automatically indent document tags as you type in order to give visual clues about the structure of the document instance. Just as edits to existing text can throw off the justification so that you need `sgml-fill-element` to line it up again, the indenting can get mangled by edits as well, and there's a command to fix it: `indent-region`. Being a regular Emacs command (as opposed to a special PSGML command) the **`M-C-\`** keystroke that invokes this doesn't begin with a **`C-c`**.

It indents according to SGML element structure because it follows the rules set by the current mode, which in this case is PSGML.

For example, the following text, after some edits, has haphazard indentation:

```
    <sect2>
 <title>A Bosom Friend</title>
     <para>We then turned over the book together, and I endeavored to
      explain to him the purpose of the printing, and the meaning of the few
      pictures that were in it.</para>
 <figure>
<title>A Sample Figure</title>
<graphic fileref="giftest.gif" format="gif"></graphic>
    </figure>
  <para>Thus I soon engaged his interest; and from that we went to
jabbering the best we could about the various outer sights to be seen
in this famous town.</para>
```

After we mark it as a region and press **M-C-**, PSGML reindents it to make the element structure much clearer:

```
  <sect2>
    <title>A Bosom Friend</title>
    <para>We then turned over the book together, and I endeavored to
explain to him the purpose of the printing, and the meaning of the few
pictures that were in it.</para>
    <figure>
      <title>A Sample Figure</title>
      <graphic fileref="giftest.gif" format="gif"></graphic>
    </figure>
    <para>Thus I soon engaged his interest; and from that we went to
jabbering the best we could about the various outer sights to be seen
in this famous town.</para>
```

Moving Your Cursor Around an SGML Instance

In the section "Moving Your Cursor Around," we saw how to move your cursor one character or word to the left or right, one line up or down, and to the beginning or end of a line or document. PSGML takes advantage of an SGML document's struc-

ture to provide several handy new cursor movement commands. Essentially, these keystrokes let you quickly put your cursor where you want without the tags getting in your way.

To more easily remember PSGML cursor movement keystrokes, keep in mind that many are the SGML equivalents of similar Emacs commands, entered by preceding the same keystroke with the **Escape** key. (Remember that the Emacs documentation convention represents a combination like **Escape** followed by **Ctrl+A** as **C-M-a**, with the "M" indicating the **Escape** key.)

The Emacs keystrokes that move the cursor to the beginning and end of the current line are **C-a** and **C-e**, respectively. The PSGML keystroke to move the cursor to the beginning of the live element is **C-M-a**, (`sgml-beginning-of-element`) which moves it to the first data character of the current element. **C-M-e** (`sgml-end-of-element`) moves to the end of the live element, putting the cursor at the "<" that begins the element's end-tag so that you can add new text to the end of that element. (Selecting **Beginning of element** or **End of element** from the **Move** menu also executes these commands.)

For example, with your cursor on the "i" in "shirt" in Fig. 2.29, pressing **C-M-a** moves it to the "s" in the same word. **C-M-e** moves it right after the "t" that ends the word, because the entire `emphasis` element consists of the word "shirt." With your cursor on the word "two" following "shirt," **C-M-a** moves it to the "I" that starts the first sentence, and **C-M-e** moves it right after the period following the word "Pacific" because that `para` element is the new live element once your cursor is outside of that `emphasis` element.

Pressing either of these keystrokes a second time won't have any effect, just as pressing **C-a** when your cursor is already at the beginning of a line won't move your cursor anywhere.

```
  <sect1>
    <title>The Carpet-Bag</title>
    <para>I stuffed a <emphasis>shirt</emphasis> or two into my
old carpet-bag, <emphasis>tucked</emphasis> it under my arm, and
<emphasis>started</emphasis> for Cape Horn and the Pacific.</para>
    <para>Quitting the good city of old Manhatto, I duly
arrive in New Bedford.</para>
    <para>It was on a Saturday night in December.</para>
  </sect1>
```

Figure 2.29. Moving your cursor around

To jump forward or backward an element, use the **C-M-f** (`sgml-forward-element`) and **C-M-b** (`sgml-backward-element`) keystrokes, or select **Forward element** or **Backward element** from the **Move** menu. `sgml-forward-element` moves to the point right after the live element's next subelement; for example, with your cursor on the word "stuffed" in Fig. 2.29, **C-M-f** jumps it to the space just before the word "or." Pressing it a second and third time moves your cursor to the space before the words "it" and "for" respectively, but pressing it a fourth time displays the message "No more elements in para element" in the minibuffer. Your cursor is still in the `para` element, but there are no more elements within it after the word "for."

For the same reason, putting your cursor on the word "shirt" and pressing **C-M-f** displays the message "No more elements in emphasis element" because there are no elements within that `emphasis` element to jump past.

`sgml-backward-element` behaves similarly in the opposite direction. With your cursor on the word "it" in Fig. 2.29's first `para` element, pressing **C-M-b** jumps it to the < of the `emphasis` start-tag following "carpet-bag, " and repeating the keystroke moves the cursor to the `emphasis` start-tag after the phrase "stuffed a " near the beginning of the paragraph. Pressing it a third time displays an error message similar to the one we saw when we tried to jump the cursor forward in an element with no more contained elements, but this time no contained elements precede the cursor: "No previous element in para element."

`sgml-forward-element` and `sgml-backward-element` demonstrate PSGML's intelligence because they show that PSGML knows your document is not just a bunch of text with tags mixed in but rather an organized document with elements inside of other elements in a specific structure. Three other commands that take advantage of your document's structure to help you navigate it are `sgml-down-element`, `sgml-up-element`, and `sgml-up-backward-element`. (In addition to the keystrokes described below, these are available as **Up element**, **Down element**, and **Backward up element** on the **Move** menu.)

It's easy to think of "up" and "down" as moving up and down on your screen, but that's what the "forward" and "backward" commands do. These "up" and "down" commands move up and down in your element hierarchy, moving up to an element's container element or down into one of its contained elements.

For example, with your cursor after the `title` end-tag in Fig. 2.29, your cursor is in the `sect1` element but not in any of its contained elements. Pressing **`C-M-d`** for the `sgml-down-element` command jumps your cursor to the beginning of the first element in `sect1` after the cursor's current position: the "I" beginning the "I stuffed a" phrase. Pressing **`C-M-d`** again moves it to the beginning of the first element inside of that `para` element: the "s" in the `emphasis` element's "shirt." Pressing it again displays an error message in the minibuffer because this `emphasis` element only has data, with no subelements.

When moving up the element hierarchy, you have a choice of moving your cursor forward or backward to a point in the live element's parent element. Starting at the `emphasis` element with the word "shirt," the next element up the hierarchy is a `para`, so pressing **`C-c C-n`** for the `sgml-up-element` command jumps the cursor to a position in that `para` element right after the `emphasis` element just before the word "or." Pressing **`C-c C-n`** again moves it one level up from that `para` element to a point after the `para` end-tag following the words

"Cape Horn and the Pacific." (Again, don't forget the up/down forward/backward distinction, or this command may seem counter-intuitive because your cursor goes down the screen as you move up the document hierarchy.) At this point, the lowest level element containing the cursor's position is a `sect1`, so you've moved the cursor up from an `emphasis` element to a `para` element to a `sect1` element.

To move up the hierarchy to an earlier point in the parent element instead of a later point, press **`C-M-u`** for the `sgml-backward-up-element` command. With your cursor on the word "shirt" in Fig 2.29's first `emphasis` element, this puts your cursor after the words "stuffed a " at the "<" that begins that `emphasis` element's start-tag. Now it's in a `para` element; pressing this keystroke again moves the cursor to the "<" of the `para` element's start-tag preceding the "I stuffed a" sentence. Repeating this keystroke moves it to the beginning of the `sect1` start-tag, the `chapter` start-tag, and so on up to the beginning of your document element.

Another great way to quickly put your cursor where you want it is the `sgml-next-data-field` keystroke (**`C-c C-d`**). It jumps your cursor to the next valid place for character data, regardless of the structural relationship of the tags around the cursor. For example, whether your cursor is on or before the `sect1` or `title` start tags in Fig. 2.30, pressing **`C-c C-d`** moves it to the "T" beginning the phrase "The Carpet-Bag." Pressing it again jumps to the "I" in "I stuffed," then the "s" in shirt, and then the space before the word "or." Selecting **Next Data Field** from the **Move** menu also executes this command, although you'll use `sgml-next-data-field` often enough that the keystroke is more efficient.

```
  <sect1>
    <title>The Carpet-Bag</title>
    <para>I stuffed a <emphasis>shirt</emphasis> or two into my
old carpet-bag, <emphasis>tucked</emphasis> it under my arm, and
<emphasis>started</emphasis> for Cape Horn and the Pacific.</para>
```

Figure 2.30. Using `sgml-next-data-field`

After it inserts the tags, the `sgml-insert-element` keystroke (**`C-c C-e`**) usually puts your cursor right where you want to start typing, but sometimes you find your cursor surrounded by tags representing enough elements within elements that the best place for your cursor isn't immediately obvious. (This is, after all, SGML.) When you finish inserting and arranging the tags that show part of a document's structure, **`C-c C-d`** is often the quickest way to go back to typing your document's data content.

Deleting, Moving, and Copying Elements

We already saw that the Emacs `kill-region` keystroke (**`C-w`**), like the "cut" command in other text editors and word processors, deletes the marked area (or, in Emacs talk, the "region") from your document and stores it in a temporary buffer, available for pasting ("yanking") to a new location with the **`C-y`** keystroke if you plan to move the text instead of deleting it.

To kill an SGML element, you could move the cursor to the element's beginning, set the mark there, move to the end, and press **`C-w`**, but PSGML gives you the `sgml-kill-element` keystroke (**`C-M-k`**) (and the **Kill Element** choice of the **Modify** menu) to automate this. It kills the text from the cursor's current location to the end of the next element contained by the live element.

Because it also deletes any text from the cursor to the start of the element you meant to delete, be careful where you put your cursor before you press **`C-M-k`**. (And, remember the **`C-_`** keystroke to `undo` mistakes when you use the commands described in this section.) For example, with your cursor on the "n" of the word "native" in Fig. 2.31, pressing **`C-M-k`** deletes the phrase "native of <emphasis>Kokovoko</emphasis>"

```
<sect1>
  <title>Biographical</title>
  <para>Queequeg was a native of <emphasis>Kokovoko</emphasis>, an
  island far away to the West and South.</para>
  <para>It is not down in any map; true places never are.</para>
</sect1>
<sect1>
  <title>Wheelbarrow</title>
  <para>Next morning, Monday, after disposing of the embalmed head
  to a barber, for a block, I settled my own and comrade's bill;
  using however, my comrade's money.</para>
</sect1>
```

Figure 2.31. Using `sgml-kill-element`

As with the `sgml-forward-element` keystroke (**`C-M-f`**), killing the next subelement with **`C-M-k`** displays an error message in the minibuffer if there is no next subelement. For example, pressing it with your cursor on the word "island" in the last example displays the message "No more elements in para element" because the cursor is in a `para` that has no subelements after the cursor's position. In fact, "sgml-kill-next-subelement" might have been a better name for the **`C-M-k`** keystroke's command, but that's a bit long.

`sgml-kill-element` is handy for moving elements because it treats the entire element as a structural unit that includes its subelements. If you move an element that has subelements which in turn have subelements, `sgml-kill-element` kills them all into the kill buffer. In Fig. 2.31, with your cursor just before the `sect1` start-tag preceding the "Biographical" title, pressing **`C-M-k`** grabs everything down to the `sect1` end-tag after the phrase "true places never are," with the result shown in Fig. 2.32.

```
<sect1>
  <title>Wheelbarrow</title>
  <para>Next morning, Monday, after disposing of the embalmed head
  to a barber, for a block, I settled my own and comrade's bill;
  using however, my comrade's money.</para>
```

Figure 2.32. After killing the first `sect1` element

Moving the cursor after the remaining `sect1` end-tag and pressing the Emacs `yank` keystroke (**`C-y`**) copies the recently killed `sect1` element titled "Biographical" and all of its subelement from the kill buffer to the cursor's position. This moves it after the `sect1` element that it used to precede (see Fig. 2.33).

```
<sect1>
  <title>Wheelbarrow</title>
  <para>Next morning, Monday, after disposing of the embalmed head
  to a barber, for a block, I settled my own and comrade's bill;
  using however, my comrade's money.</para>
</sect1>
<sect1>
  <title>Biographical</title>
  <para>Queequeg was a native of <emphasis>Kokovoko</emphasis>, an
  island far away to the West and South.</para>
  <para>It is not down in any map; true places never are.</para>
</sect1>
```

Figure 2.33. First `sect1` element yanked to a new location

Before we move on to copying elements, two other PSGML deletion commands are worth mentioning. They don't delete entire elements, but both speed up common operations enough to come in handy.

The `sgml-untag-element` keystroke (**`C-c -`**) removes the start- and end-tags from the live element. With your cursor on the word "Queequeg" in Fig. 2.33, **`C-c -`** would remove the `para` start- and end-tags; with your cursor just before that `para` start-tag, it would remove the `sect1` start- and end-tags. Selecting **Untag Element** from the **Modify** menu also performs this command.

This command's greatest value is in removing tags around inline elements (that is, elements in mixed content). For example, if you decide that the word "Kokovoko" in that example shouldn't be emphasized, putting your cursor anywhere on that word and pressing this keystroke removes the `emphasis` start- and end-tags.

The `sgml-kill-markup` keystroke (**`C-c C-k`**) and the **Kill Markup** choice of the **Modify** menu kill anything between the "<" at the cursor and its matching ">" character. This includes

markup declarations (such as comments and marked section delimiters), tags, and processing instructions. This command is great for deleting the comments that PSGML inserts to give you hints about necessary elements when you have `sgml-auto-insert-required-elements` set to a non-nil value, because after you've followed the comment's advice, it only clutters up your document.

PSGML has no built-in command to delete from the cursor position to the end of the current element, so I wrote a macro to do this for my `.emacs` file. It didn't work, so I'd like to thank Lennart Staflin, PSGML's author, for straightening it out for me. Once you add the following to your `.emacs` file, pressing **`C-c k`** calls the macro.

```
(defun sgml-kill-to-eoelement ()  ; kill to end of element
  (interactive)
  (let ((start (point)))
  (sgml-end-of-element)
  (kill-region start (point))))

; assign to ^Ck keystrokes
(define-key global-map "^Ck" 'sgml-kill-to-eoelement)
```

Copying Elements

To copy an element with all of its subelements, you have two options. First, you could delete it using any of the keystroke sequences just described, immediately paste it back at the cursor's position, and then paste it to any place where you want a new copy.

Or, to copy it to the kill buffer without deleting it, you can perform the following steps:

1. Press the `sgml-backward-up-element` keystroke (**`C-M-u`**) to move your cursor to the beginning of the element's start-tag. If the cursor is in a subelement of the element that you intend to copy, you may need to press this more than once to put the cursor where you want it.
2. Press **`C-@`** or **`C-space`** to mark that point as the beginning of a region to copy.
3. Press the `sgml-forward-element` keystroke (**`C-M-f`**) to move your cursor after the element's end-tag.
4. Press the regular Emacs `kill-ring-save` keystroke (**`M-w`**) to copy that element into the kill ring.

To summarize: **`C-M-u`**, **`C-@`**, **`C-M-f`**, and **`M-w`**. Once you get used to this, you'll find it much quicker than the "manual" way to copy an element into a kill ring or word processor clipboard, where you would enter a command to search backward, find the start-tag, mark that point as the beginning of the block to copy, enter the command to search for the end-tag, and then copy the region to the clipboard or kill ring.

To copy an element into the kill ring even more quickly, add the following macro and key definition to your `.emacs` file and press **`C-c w`** to copy the current element:

```
(defun sgml-copy-element ()
  (interactive)
  (sgml-backward-up-element)
  (let ((start (point)))
  (sgml-forward-element)
  (kill-ring-save start (point))))

; assign to ^Cw keystroke
(define-key global-map "^Cw" 'sgml-copy-element)
```

(I'd like to thank Lennart Staflin for helping me with this macro as well.)

Editing Attributes

Including the PSGML menus, there are four ways to edit an element's attribute values. The first is to simply move your cursor to the element's start-tag and type in the attribute name and value. For example, to add an `HREF` value so that the `a` element in Fig. 2.34 links to a Nantucket web site,

```
<p>Now, the Pequod had sailed from <a>Nantucket</a> at the
very beginning of the Season-on-the-Line. </p>
```

Figure 2.34. Adding an attribute value by typing it

you could move your cursor to the ">" in the start-tag and just type in the attribute and value as shown in Fig. 2.35.

```
<p>Now, the Pequod had sailed from <a href="http://www.nantucket.com">Nantucket\
</a> at the
very beginning of the Season-on-the-Line.</p>
```

Figure 2.35. Entered attribute value

This is no fun, and could be done with any text editor. The other methods for editing attribute values take advantage of PSGML features that make this easier.

The PSGML `sgml-insert-attribute` keystroke (**`C-c +`**) and the **Insert Attribute** choice of the **Markup** menu are more versatile than their names suggest because they also let you edit existing attribute values. With your cursor on the start-tag, end-tag, or data content of an element, pressing this keystroke displays the prompt `Attribute name:` in the minibuffer. You can take advantage of Emacs's completion feature, so you only need to type the first few characters and then press **`Tab`** or **`space`** to finish the attribute name. If you press one of these keys without specifying enough letters to indicate which attribute value to edit, PSGML displays the possibilities in a

separate window. For example, if you press **Tab** without first typing any letters of the a element's attribute names, you'll see the attribute name list shown in Fig. 2.36.

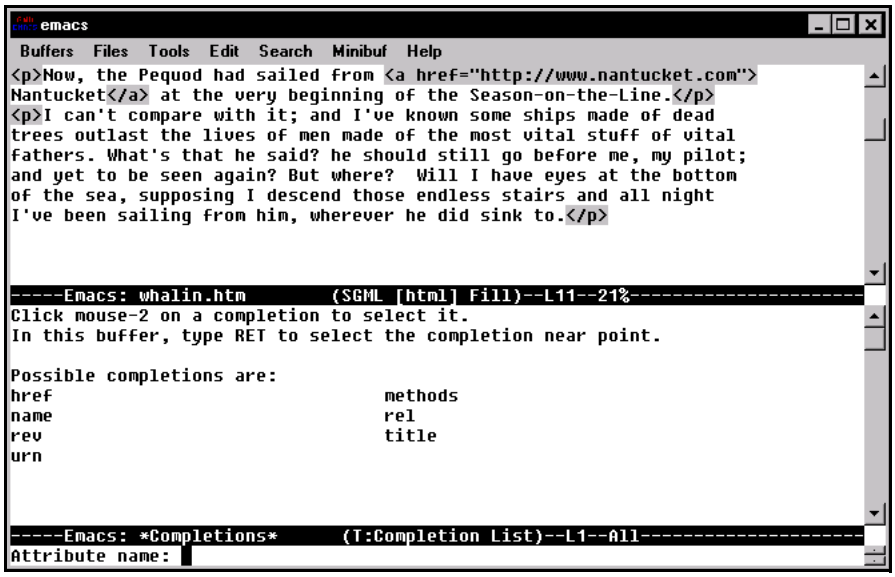

Figure 2.36. Listing the a element's attributes with the `sgml-insert-attribute` command

After you enter the attribute name and press **Enter**, the minibuffer displays the attribute's declared value (data type) and current specified value (if any) and asks you to enter the new value as shown in Fig. 2.37.

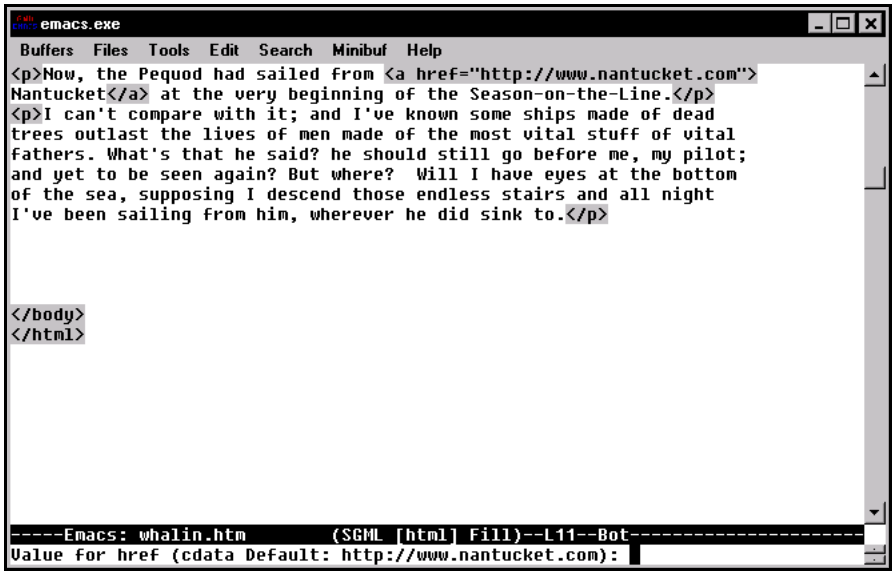

Figure 2.37. The `sgml-insert-attribute` command's prompt for a new `href` value

You don't need to enter quotes around the value; PSGML adds them for you in the document instance. Responding to the prompt above with `http://www.nantucket.gov`, as shown in Fig. 2.38,

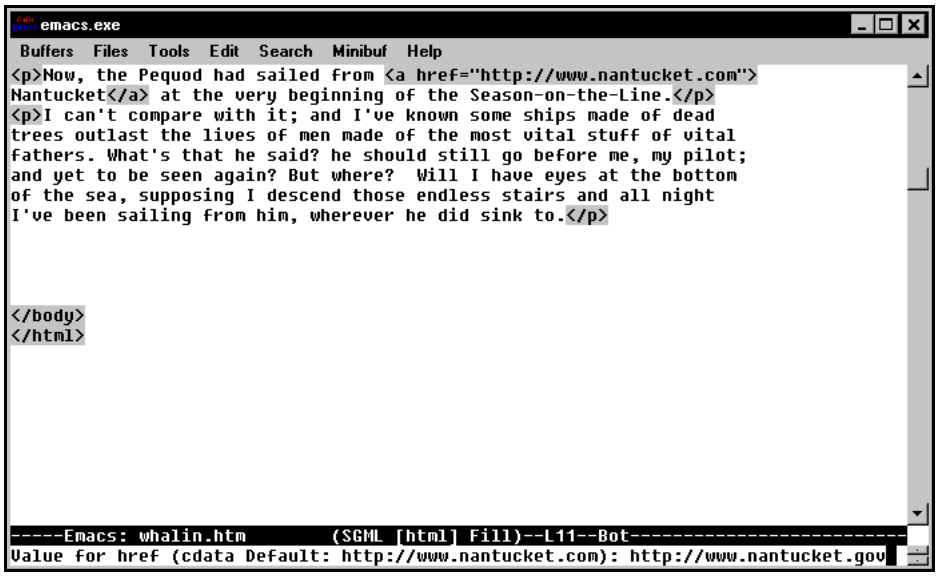

Figure 2.38. Entering a new `href` value

and pressing **`Enter`** changes the markup to that shown in Fig. 2.39.

```
<p>Now, the Pequod had sailed from <a href="http://www.nantucket.gov"></a> at
the very beginning of the Season-on-the-Line. </p>
```

Figure 2.39. Result of editing the attribute value

You can even use completion to enter the attribute's value, if the declared value is a name token group. For example, the HTML DTD's `IMG` element defines its `ALIGN` attribute with this line:

```
ALIGN (top|middle|bottom) #IMPLIED
```

When editing an `IMG` element's `ALIGN` attribute, after you press **`C-c +`** and enter `align` as the attribute name, pressing either **`t`**, **`m`**, or **`b`** (for "top," "middle," or "bottom") and then **`Tab`** enters the complete attribute value for you.

An even easier way to edit attributes, especially when editing more than one from the same element, is by using the `sgml-edit-attributes` keystroke (**`C-c C-a`**) or by selecting **Edit Attributes** from the **Modify** menu. This splits the screen to display a new window, showing the live element's attributes listed on a form that you fill out. (As with `sgml-insert-element`, your cursor can be on the element's start-tag, end-tag, or data content when you invoke this command.) Pressing **`C-c C-a`** with your cursor on the "Nantucket" text displays the split screen shown in Fig. 2.40.

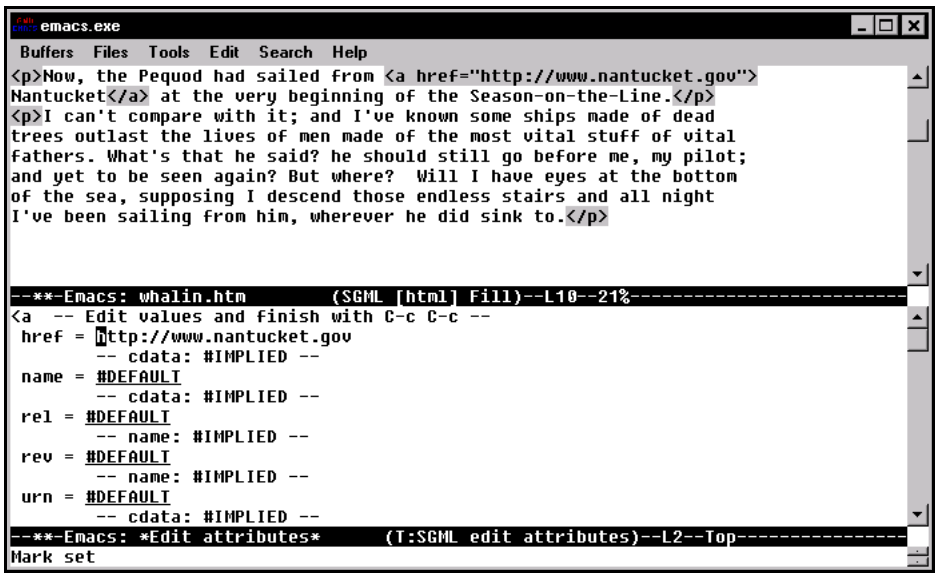

Figure 2.40. Entering attribute values with the `sgml-edit-attributes` "form"

Move your cursor to any attribute value and enter the new value. All your cursor keys work here, and the **`Tab`** key provides a shortcut for jumping from one field to the next.

As the window tells you, pressing **C-c C-c** indicates that you are finished inserting your new values into the document markup and closes the new window. To abort the attribute window, the usual Emacs `delete-window` keystroke (**C-x 0**) closes that window and ignores any changes made there.

Completion doesn't work when entering attribute values, but you don't really need it because the attribute editing window lists the possibilities for any element whose valid choices are listed in the attribute definition. For example, Fig. 2.41 shows the attribute editing window for an HTML `IMG` element; note the "bottom middle top" list on the second line of the `ALIGN` entry:

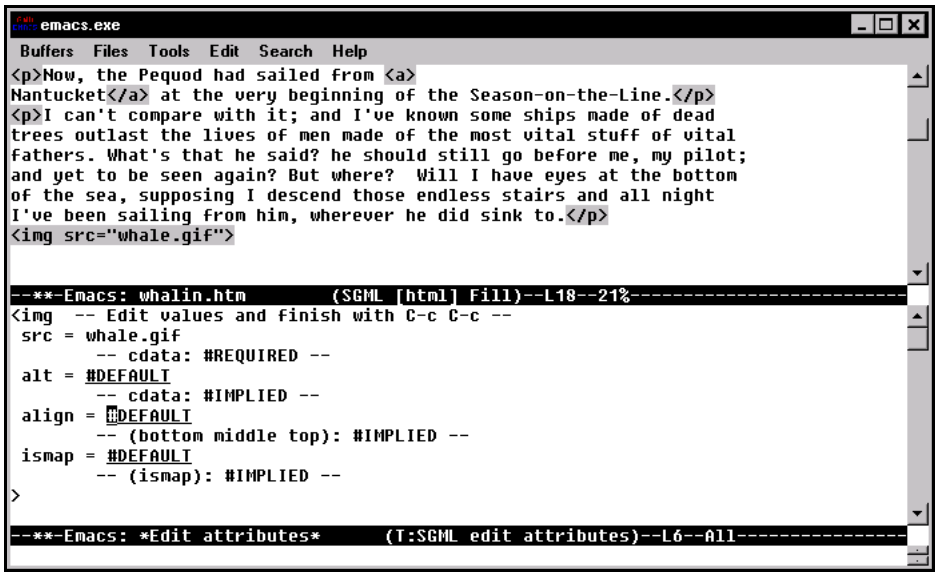

Figure 2.41. Attribute editing form showing possible `img` values

The fourth and easiest way to edit attribute values is by selecting **Insert Attribute** from the **Markup** menu if menus are available. Don't be misled by its name—like the `sgml-insert-attribute` command automated by the **C-c +** keystroke, this menu choice lets you edit existing attribute values

as easily as inserting new ones. It displays a menu of the live element's attributes, and selecting any of those attributes displays the available choices. If the attribute's declared value is a named token group, PSGML displays each of the choices as a choice on this menu so that you merely need to click the value to insert it into the document instance markup (see Fig. 2.42).

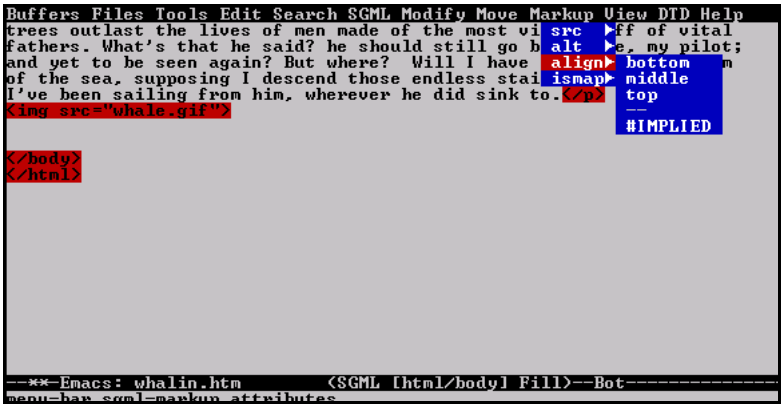

Figure 2.42. Selecting attribute values from a menu

Most of these attribute value menus include **Set attribute value** as a choice. Selecting it displays a prompt in the minibuffer that waits for you to enter a value.

Finding Tagging Mistakes

We saw that, after first displaying a document instance in Emacs, the `sgml-next-trouble-spot` keystroke (**`C-c C-o`**) is the quickest way to parse the text to read its tags so that PSGML knows the document's structure. For many PSGML users, it's the first thing they do when they bring up a document.

It's handy to repeat this keystroke several times as you edit a document, just like saving it. (In fact, doing it just before each save helps to ensure that you're saving a structurally sound document.) Move your cursor to the top of your document or

before your most recently edited block of text and press **C-c C-o**. If your cursor jumps to the bottom and the message "Ok" displays in your minibuffer, PSGML found no problems.

Remember, however, that PSGML is not a full validating parser; it just helps you along. To validate properly, the `sgml-validate` keystroke (**C-c C-v**) splits your screen into two windows, runs a parser against your document, and puts error and status messages in the new window. The default parser in PSGML 0.4 beta 2 is James Clark's free nsgmls parser, which is covered in more detail in the "Parsing and Validating SGML Documents with nsgmls" chapter; we'll see below how to set PSGML to run another parser if you prefer.

After you press **C-c C-v** (or select **Validate** from the **SGML** menu), PSGML first displays the command it's about to run in the minibuffer to give you the opportunity of editing it:

```
Validate command: nsgmls -s whalin.sgm
```

(The `s` switch tells nsgmls to only show error messages and not to show the output of the parse.) The edit you're most likely to make is to insert the name of a file with the document's SGML declaration before the document's filename. nsgmls, being a very thorough validator and parser, cares much more about the SGML declaration than PSGML does and assumes the Reference Concrete Syntax if it doesn't find one. Because this concrete syntax won't allow element type names longer than eight characters, it spits out a two-line error message for each one it finds in your DTD and document instance, and for an instance of more than a couple of lines, that's a lot of error messages.

When the validation command pops up in your minibuffer, your cursor will be there, so move it to the appropriate place and enter the name of a file with the appropriate SGML declaration:

```
Validate command: nsgmls -s ..\catalog\docbook.dcl whalin.sgm
```

PSGML assumes that the parsing program is in your path. If it can't find it, you'll see an error message in the output window.

Validator Output

When you press **`Enter`** to accept the validation command in your minibuffer, PSGML splits your screen to create a new window with a buffer titled `*sgml validation*`. The command executing the validator appears in that window and the validation program goes to work, as shown in Fig. 2.43.

If the word "done" eventually appears at the end of the minibuffer message and you never see anything else happen, then congratulations—the validator found nothing wrong with your document.

If the validator does find errors, it lists them in the validation message window. PSGML makes it easy to sort through the messages and quickly find the problems that they identify in the document. Let's add two errors to an otherwise valid file and see what the error messages look like.

The following shows the beginning of a document that conforms to the DocBook DTD, with two exceptions: I put a stray `sect1` start-tag between the `emphasis` tags around the word "shirt" and I gave the second `para` element a `hair` attribute it doesn't have, with an attribute value of "red."

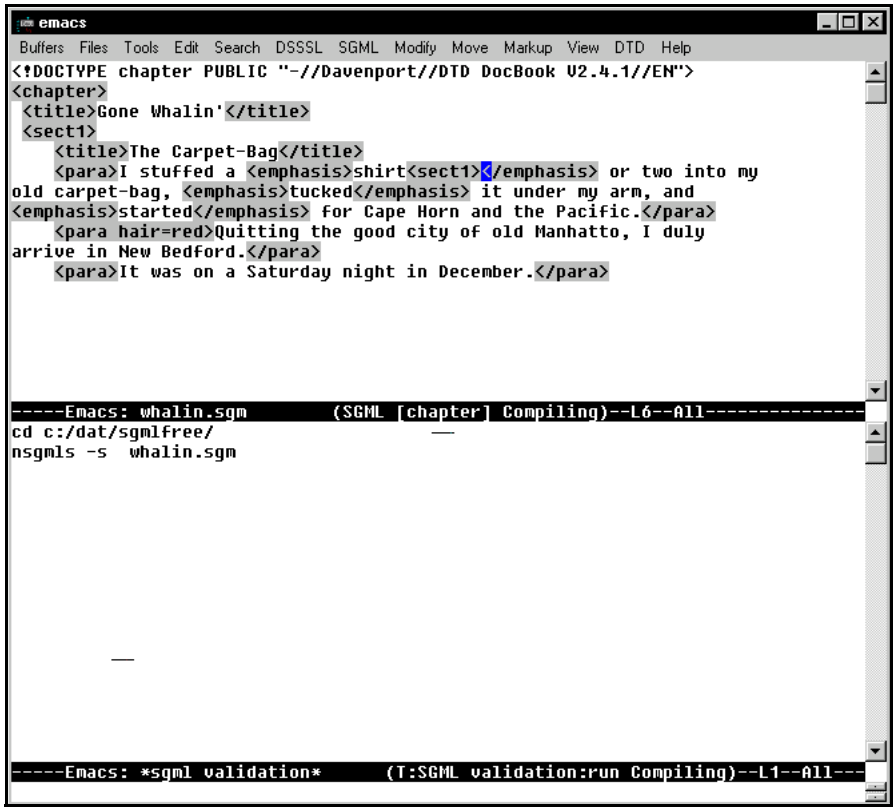

Figure 2.43. `sgml-validate` begins validating the document

```
<!DOCTYPE chapter PUBLIC "-//Davenport//DTD DocBook V2.4.1//EN">
<chapter>
 <title>Gone Whalin’</title>
 <sect1>
    <title>The Carpet-Bag</title>
    <para>I stuffed a <emphasis>shirt<sect1></emphasis> or two into my
old carpet-bag, <emphasis>tucked</emphasis> it under my arm, and
<emphasis>started</emphasis> for Cape Horn and the Pacific.</para>
    <para hair=red>Quitting the good city of old Manhatto, I duly
arrive in New Bedford.</para>
    <para>It was on a Saturday night in December.</para>
```

After pressing **C-c C-v** and sending this document instance to nsgmls, the following error messages appear in the validation window:

```
nsgmls:whalin.sgm:6:43:E: document type does not allow element "SECT1" here
nsgmls:whalin.sgm:6:54:E: "SECT1" not finished but containing element ended
nsgmls:whalin.sgm:6:54:E: end tag for "SECT1" omitted, but OMITTAG NO was
specified
nsgmls:whalin.sgm:6:37: start tag was here
nsgmls:whalin.sgm:9:15:E: there is no attribute "HAIR"
nsgmls:whalin.sgm:11:57:E: end tag for "SECT1" omitted, but OMITTAG NO was
specified
nsgmls:whalin.sgm:4:1: start tag was here
nsgmls:whalin.sgm:11:57:E: end tag for "CHAPTER" omitted, but OMITTAG NO was
specified
nsgmls:whalin.sgm:2:0: start tag was here
```

These messages have a couple of things in common with programming-related error messages, both to our advantage:

- The ripple effect of one error causing many error messages means that you shouldn't panic when you see a lot of error messages. We already know that the document instance only has two problems, so the next time you see eight error messages, maybe they're the result of only two problems.
- Emacs has a special command, automated by a keystroke, that quickly finds the text line that caused each error message. We can use this to locate SGML errors.

The Emacs `next-error` keystroke (**C-x `**) is not a special PSGML command, which is why it doesn't begin with **C-c**. Programmers use it to find the lines in their source code that caused each compiler error message. It jumps the cursor in your document instance window to the line with the next error and lines up the first error message at the top of the `*sgml validation*` window (see Fig. 2.44).

The extraneous `sect1` tag caused five error messages and the `hair` "attribute" caused one. Error message lines without a one-letter code after the second number provide background information on the previous error; for example, the line after the one about the missing `sect1` end tag tells us that the `sect1` start tag was on character 37 of line 6.

The "Parsing and Validating SGML Documents with nsgmls" chapter describes nsgmls and sgmls messages in more detail.

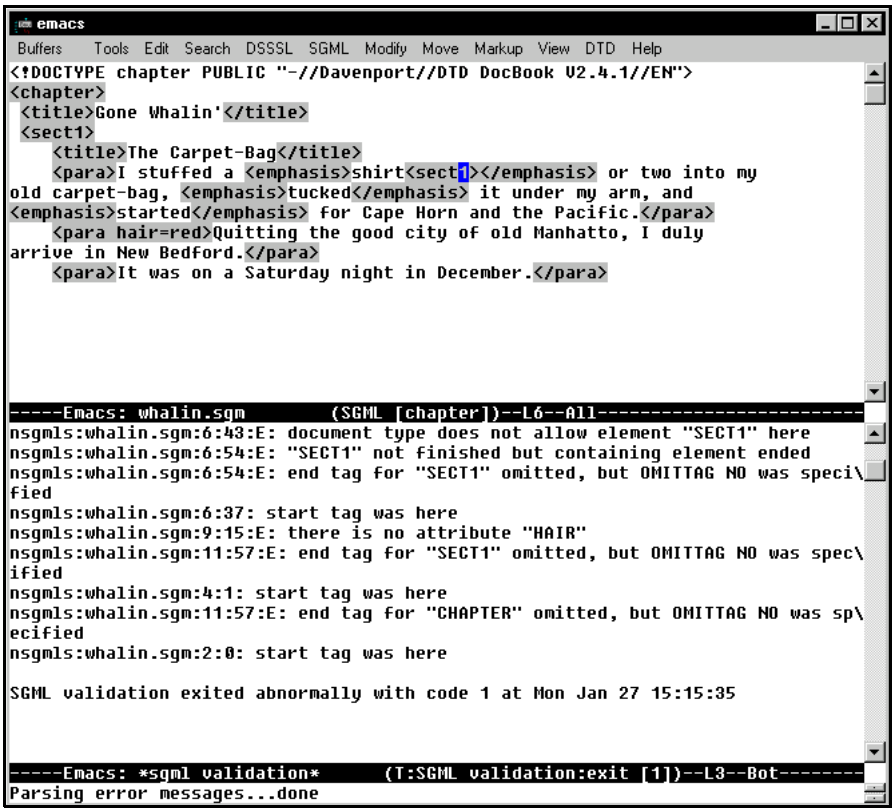

Figure 2.44. First error of output

When multiple error messages refer to the same line, press **C-x '** to skip to the first error message of the next document line that has an error, and the document window cursor jumps to that line. (I suppose `first-error-message-of-next-source-line-with-error`, while more accurate, would be too wordy compared with `next-error`.) With our example, pressing this keystroke three more times puts the document window cursor at the ninth line and lines up the error message about that line at the top of the validation window (see Fig. 2.45).

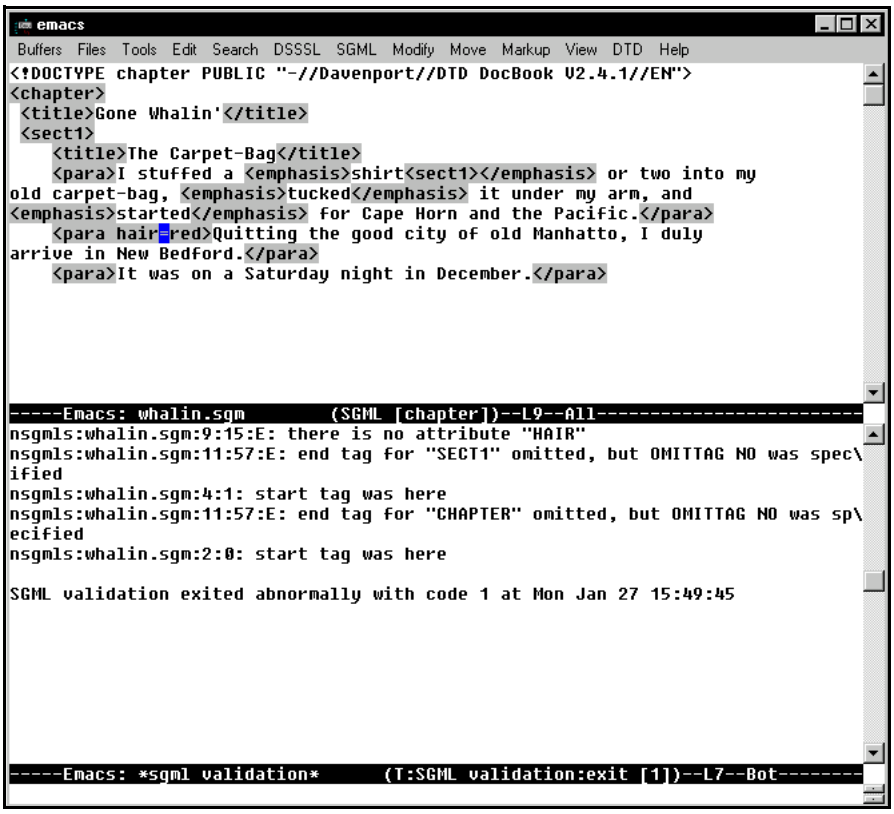

Figure 2.45. Second error of output

If you press **C-x '** when there are no more errors, a minibuffer error message tells you "No more errors."

Changing the Validation Command

PSGML creates a variable in the Emacs environment called `sgml-validate-command` to store the command it executes when you press **C-c C-v**. If you want to change this command, you can set it to a different value in your `.emacs` file. For example, let's say the Emacs `describe-variable` keystroke (**C-h v**) shows that the `sgml-validate-command` variable has the following value:

```
nsgmls -s %s %s
```

(The `%s %s` shows where Emacs substitutes the name of the file being edited when it is executed. The syntax will be familiar to C programmers.) If you want PSGML to use the `sgmls` program instead, the following line in your `.emacs` file sets the variable to run `sgmls` when you enter **C-c C-v**:

```
(setq sgml-validate-command "sgmls -s %s %s")
```

Remember how the validation command might need an SGML declaration file? If you often use the same SGML declaration file, you can add it to the `.emacs` line that sets the `sgml-validate-command` variable so that you don't have to specify it every time you validate a file:

```
(setq sgml-validate-command "sgmls -s \\sgml\\dtds\\docbook.dcl %s %s")
```

(Note the use of double backslashes to put literal backslashes in the string.) Even if you add an SGML declaration file name to the default command, you don't have to use that declaration every time. After you press **C-c C-v**, you still have a chance to edit the validation command before PSGML executes it.

Other Handy PSGML Features

PSGML has too many features and too many ways to take advantage of them for this chapter to cover them all. Still, several more valuable tricks are worth mentioning.

Displaying Markup in Different Fonts and Colors

When you use a terminal (or terminal emulation program) and a version of Emacs that can show text in different fonts or colors, PSGML can take advantage of this to make it easier to distinguish between data content and different kinds of markup. Its default settings display SGML comments in italics; entity references in bold italics; and tags, processing instructions, the SGML and DOCTYPE declarations, and short references in bold text.

If your version of Emacs can show different fonts, you're probably using a UNIX version with an XWindows terminal. Other versions typically substitute colors for these fonts; for example, The EMX version of GNU Emacs 19.29.2 shows comments in red, entity references in yellow, and tags and other markup in white text against its default gray background.

If you've issued the `sgml-next-trouble-spot` command and you still don't see any fonts or colors, you can tell Emacs and PSGML to display markup in your choice of colors by adding a few lines to your `.emacs` file. These lines must perform four basic steps:

1. Create "faces" to assign to the markup categories.
2. Assign attributes to the faces.
3. Assign faces to markup categories.
4. Set the `sgml-set-face` variable to a non-nil value so that PSGML knows to pay attention to the face settings.

The following `.emacs` code demonstrates how to assign colors to comments, tags, and entity references.

```
;;;;; Assign colors to markup. ;;;;;

; Create faces to assign to markup categories.
(make-face 'sgml-comment-face)
(make-face 'sgml-start-tag-face)
(make-face 'sgml-end-tag-face)
(make-face 'sgml-entity-face)

; Assign attributes to faces. Background of white assumed.
(set-face-foreground 'sgml-comment-face "White")  ; Comments: white on
(set-face-background 'sgml-comment-face "Gray")   ; gray.
(set-face-background 'sgml-start-tag-face "Gray") ; Tags: black (default)
(set-face-background 'sgml-end-tag-face "Gray")   ; on gray.
(set-face-foreground 'sgml-entity-face "White")   ; Entity references:
(set-face-background 'sgml-entity-face "Black")   ; white on black.

; Assign faces to markup categories.
(setq sgml-markup-faces
   '((comment   . sgml-comment-face)
     (start-tag . sgml-start-tag-face)
     (end-tag . sgml-end-tag-face)
     (entity . sgml-entity-face)))

; Tell PSGML to pay attention to face settings.
(setq sgml-set-face t)
```

A couple of things to note:

- To list the available colors, enter the Emacs command `list-colors-display` at the **`M-x`** command prompt. The choice varies depending on your version of Emacs.
- My choices don't seem very colorful because I picked settings that make this chapter's illustrations easier to reproduce on the printed page.
- The above example sets the start- and end-tags to the same color, but this is not required.
- You can set both foreground and background colors if you wish. I left the tag and comment backgrounds and the entity foreground color (that is, the actual text color) at the default settings.

I only set the `comment`, `start-tag`, `end-tag`, and entity appearances in the `setq sgml-markup-faces` part, but you can also set the colors of `doctype` declarations, `ignored` marked sections, `ms-start` for the start and `ms-end` for end of non-ignored marked sections, `pi` for processing instructions, `sgml` for the SGML declaration, and `shortref` for short references. Just remember to create faces for them and set the faces' attributes before the final step.

Normalizing Document Instances

The `sgml-normalize` command expands any empty tags and fills in omitted tags in a document instance. In other words, it makes sure that every non-empty element has a beginning and end-tag, whether or not the DTD requires it.

Much SGML software requires normalized SGML, so this command is a quick way to prepare your instance for use with such programs. This command is not assigned to a keystroke because you rarely need it more than once or twice in an editing session, so run it by selecting **Normalize** from the **Modify** menu or by entering `sgml-normalize` at the minibuffer **`M-x`** prompt.

Help Entering Entity References and Markup Declarations

The `sgml-complete` keystroke (**`M-Tab`**) tries to help you complete the entity reference, markup declaration, tag, or even data content word that you are entering when you press it. We've already seen how PSGML can help you enter element type names; **`M-Tab`** can only help you enter content text words if you have the `ispell` program available (an Emacs spell-checking utility—DOS/Windows Emacs users should see `http://cat.rpi.edu/~tibbetts/ispell_port.html` for a special version) but the entity reference and markup completion capabilities can be valuable.

PSGML knows that you're entering an entity reference when you've entered an ampersand and haven't entered a space following it yet. If you press **M-Tab** right after entering the ampersand, it lists all the valid entities whose name you might be entering in a separate window. If you follow the ampersand with enough letters to let PSGML narrow its guess down to one, it fills in that entity reference for you. For example, in Fig. 2.46, let's say you only typed the "&md" just before the `emphasis` start tag.

```
      <para>It was on a Saturday night in December.</para>
      <para>Unconsciously clapping the vinegar-cruet to one side of
her nose, she ruminated for an instant; then
exclaimed&md'<emphasis>No!</emphasis> I haven't seen it since I put it
there.'</para>
```

Figure 2.46. Entering the beginning of an entity reference

If you then press **M-Tab** and `mdash` is the only declared entity beginning with the letters "md," PSGML enters the "ash" for you, as shown in Fig. 2.47.

```
      <para>It was on a Saturday night in December.</para>
      <para>Unconsciously clapping the vinegar-cruet to one side of
her nose, she ruminated for an instant; then
exclaimed&mdash'<emphasis>No!</emphasis> I haven't seen it since I put it
there.'</para>
```

Figure 2.47. Completing the entity reference with **M-Tab**

Note that you must still enter the semicolon to complete the entity reference.

The `sgml-complete` command completes a markup declaration, which you're more likely to use in a DTD than in a document instance. For example, entering "<!" with no letters to give a clue about which declaration you want and then pressing **M-Tab** opens up a new window and displays the following list of possible completions:

```
sgml                            doctype
element                         entity
usemap                          shortref
notation                        attlist
uselink                         linktype
link                            idlink
```

Showing the Live Element's Context

In the section "PSGML Startup Variables," we saw that setting the Emacs variable `sgml-live-element-indicator` to a non-nil value tells PSGML to display the document type and the current live element on the mode line. To learn more about the role of the live element in the document's structure, the `sgml-show-context` keystroke (**`C-c C-c`**) lists the live element's ancestry from itself to the doctype, showing the live element's parent, that element's parent, and so forth all the way up to the top (that is, to the document element).

```
<sect1>
  <title>Biographical</title>
  <para>Queequeg was a native of <emphasis>Kokovoko</emphasis>, an
island far away to the West and South.</para>
```

For example, with your cursor on the word "Kokovoko" in the text above, pressing **`C-c C-c`** displays the following in the minibuffer:

```
#PCDATA in emphasis in para in sect1 in chapter
```

Selecting **Show Context** from the **SGML** menu also does this.

Quick Reference of Emacs and PSGML Keystrokes

Keep the following list handy as you get used to Emacs and PSGML keystrokes. You'll find a copy of it in the file `psgml-qref.txt` on the enclosed CD-ROM.

Table 2-1. Moving Your Cursor Around: Regular Emacs Keystrokes

C-f	`forward-char`	Or, “cursor right.”
C-b	`backward-char`	Or, “cursor left.”
C-Left	`backward-word`	
C-Right	`forward-word`	
C-a	`beginning-of-line`	
C-e	`end-of-line`	
C-v	`scroll-up`	Or, “page down.”
M-v	`previous-page`	Or, “page up.”
M-b	`backward-word`	
M-f	`forward-word`	
M-g	`goto-line`	Not a regular Emacs command, but set by a line in the `.emacs` file described in this chapter.

Table 2-2. Moving Your Cursor Around: PSGML Keystrokes

C-M-a	`sgml-beginning-of-element`	First data character of current element.
C-M-e	`sgml-end-of-element`	Last data character of current element.
C-c C-d	`sgml-next-data-field`	Next place where you can enter data.
C-c C-n	`sgml-up-element`	Up in element hierarchy to next character in current element's parent element.
C-M-u	`sgml-backward-up-element`	Up in element hierarchy to beginning of current element's start-tag.

C-M-d	`sgml-down-element`	Beginning of current element's next subelement.
C-M-b	`sgml-backward-element`	Beginning of current element's previous subelement.
C-M-f	`sgml-forward-element`	Right after current element's next component element.

Table 2-3. Adding, Deleting, and Moving Text: Regular Emacs Commands

C-d	`delete-char`	
C-@	`set-mark-command`	
C-space	`set-mark-command`	
C-w	`kill-region`	Cut marked region into "clipboard" (kill ring buffer).
C-k	`kill-line`	From cursor to end of line.
C-x C-x	`exchange-point-and-mark`	Jump cursor to marked region's other boundary.
C-y	`yank`	"Paste" from kill ring buffer to cursor position.
M-d	`kill-word`	
M-i	`overwrite-mode`	A toggle. Not a regular Emacs command, but set by line in `.emacs` file.
M-q	`fill-paragraph`	Justify paragraph.
M-w	`kill-ring-save`	Copy to "clipboard" (kill ring buffer).
M-y	`yank-pop`	Replace recently yanked text with previously killed or copied text.

M-C-	`indent-region`	Indent region's lines. In PSGML mode, it indents tags to show element structure.
C-q	`quoted-insert`	Insert next entered character literally, even if it's normally part of a command keystroke.

Table 2-4. Adding and Removing SGML Markup and Elements

C-c C-e	`sgml-insert-element`	
C-c <	`sgml-insert-tag`	Insert a start-tag. Best when adding tags to existing text, as opposed to adding a new element whose content you haven't typed yet.
C-c /	`sgml-insert-end-tag`	Insert an end-tag. As with **C-c <**, most valuable when adding tags to existing text.
C-c C-r	`sgml-tag-region`	Add start- and end-tags around marked region.
C-c -	`sgml-untag-element`	Remove live element's start- and end-tags. Most useful with in-line tags.
C-c C-k	`sgml-kill-markup`	Kill a tag, comment, or other piece of markup.
C-c Enter	`sgml-split-element`	Or, "make a new element like the current one." Repeat to split higher-level elements.
C-c o	`sgml-comment`	A macro added to `.emacs` file as part of the chapter.
C-M-k	`sgml-kill-element`	Kill text from cursor to end of next subelement.

C-space **C-M-e** **C-w**		Delete remainder of current element.
M-Tab	`sgml-complete`	Complete the entity reference, markup declaration, tag, or content word at the cursor.
C-M-u **C-@** **C-M-f** **M-w**		Copy current element to kill ring.
C-c C-q	`sgml-fill-element`	Justify current element.
C-c +	`sgml-insert-attribute`	Edit current element's attribute values using prompts.
C-c C-a	`sgml-edit-attributes`	Edit current element's attribute values using a form in a separate Emacs window.

Table 2-5. Getting Help and Other Information

C-h		Display help menu.
C-h ?	`help-for-help`	Describe use of on-line help.
C-h a	`command-apropos`	List commands with a certain string in them.
C-h k	`describe key`	Describe the next key pressed after **C-h k**.
C-c C-c	`sgml-show-context`	Or, after a **C-c C-a**, end attribute editing.
C-x ‘	`next-error`	Find next error in error message window.

C-c C-o	`sgml-next-trouble-spot`	Move cursor to next potential markup problem.
C-c C-v	`sgml-validate`	Send document instance to validation program.

Table 2-6. Controlling Files, Buffers, and Windows

C-x 0	`delete-window`	Delete cursor's current window.
C-x 1	`delete-other-windows`	Make the cursor's window the only one.
C-x 2	`split-window-vertically`	Split into a top and bottom window.
C-x o	`other-window`	Repeated pressing cycles cursor through open windows and minibuffer.
C-x b	`switch-to-buffer`	Display a different buffer in cursor's current window.
C-x C-b	`list-buffers`	List open buffers in a new window.
C-x C-s	`save-buffer`	As a disk file.
C-x C-w	`write-file`	As a disk file, under a new name if you like.
C-x C-c	`save-buffers-kill-emacs`	Answer prompts about saving each current buffer, then quit Emacs.
C-x C-f	`find-file`	Open a new or existing file.
C-x i	`insert-file`	Insert an existing disk file at the current cursor position.

Table 2-7. Emergencies

C-g	`keyboard-quit`	Abort current multi-step operation.
C-_	`undo`	Undo last command.

Table 2-8. Customized Behavior

C-x (	`start-kbd-macro`	Start recording a macro.
C-x)	`end-kbd-macro`	Stop recording a macro.
C-x e	`call-last-kbd-macro`	Execute last recorded macro.
C-u (number) **C-x f**	`set-fill-column`	Set right margin to the (number) column.
M-x	`execute-extended-command`	Display Emacs command prompt in minibuffer.

Table 2-9. Searching and Replacing

C-s	`isearch-forward`	Incremental search forward.
C-r	`isearch-backward`	
M-%	`query-replace`	Prompts for target and replacement text.